HOCHSCHULE
MITTWEIDA
UNIVERSITY OF
APPLIED SCIENCES

DIPLOMARBEIT

Herr Ing.

Sergej Smoljanin

Entwicklung der Firmware eines Auslese- und Steuerungssystems für das "Large Area Medipix Based Detektor Array"

Mittweida, 2013

Fakultät Elektro- und Informationstechnik

DIPLOMARBEIT

Entwicklung der Firmware eines Auslese- und Steuerungssystems für das "Large Area Medipix Based Detektor Array"

Autor:
Herr Ing. Sergej Smoljanin

Studiengang:
Informationstechnik

Seminargruppe:
KI09wF-D

Erstprüfer:
Prof. Dr.-Ing. Thomas Beierlein

Zweitprüfer:
Dr. rer. nat. Ulrich Trunk

Einreichung:
Mittweida, 04.11.2013

Verteidigung/Bewertung:
Mittweida, 2013

Bibliografische Beschreibung:

Smoljanin, Sergej:
Entwicklung der Firmware eines Auslese- und Steuerungssystems für das "Large Area Medipix Based Detektor Array". –2013. – viii, 85, VII S. Mittweida, Hochschule Mittweida, Fakultät Elektro- und Informationstechnik, Diplomarbeit, 2013

Referat:

Die vorliegende Arbeit befasst sich mit der Entwicklung eines Detektor Steuerungssystems, welches in Form eines SoC[1] als FPGA Firmware Komponente realisiert werden soll. Das SoC soll die Steuerung und Überwachung aller Einheiten und Komponenten des Detektor-Systems gewährleisten und eine Read-Out Geschwindigkeit der Medipix3 Chips von bis zu 1000 Bildern pro Sekunde ermöglichen. Da der Detektor für unterschiedliche Experimente flexibel einsetzbar sein soll, sollen zwei unterschiedliche Ethernet-Schnittstellen zur Ansteuerung des Detektors mit dem Computer und zur Datenauslese implementiert werden. Wobei das Gigabit Ethernet Interface als Einheit des SoC realisiert werden soll, welches sowohl für die Übertragung der Detektor-Daten als auch für die Übertragung der Befehle für die Steuerung des Detektors verwendet wird. Die vom PC kommenden Steuerbefehle müssen von dem SoC empfangen, Interpretiert und Ausgeführt werden.

---

[1] System on Chip

Inhalt

Abbildungsverzeichnis

Tabellenverzeichnis

1 Einleitung

Motivation

Im Bereich der Forschung mit Photonen am Deutschen Elektronen-Synchrotron (DESY[2]) in Hamburg werden für die experimentelle Grundlagenforschung verschiedenste Detektoren zum Nachweis der Synchrotronstrahlung eingesetzt. Die seit 2009 betriebene Speicherring-Röntgenstrahlungsquelle PETRA III ist die weltweit leistungsstärkste dieser Art und stellt damit höchste Anforderungen an die neuen Röntgen-Detektoren. Zum Beispiel werden aufgrund der hohen Intensität der Röntgenstrahlung zeitlich aufgelöste Experimente möglich. Dies erfordert eine hohe Bildwiederholungsfrequenz der Detektor-Systeme und damit verbunden eine hohe Auslesegeschwindigkeit der anfallenden Datenmenge. Um die neuen Möglichkeiten ausnutzen zu können wurde die Entwicklung des „Large Area Medipix Based Detector Array“ (LAMBDA) [16] begonnen. Dabei handelt es sich um einen zweidimensionalen Röntgen-Detektor welcher auf dem, am CERN[3] entwickelten, Medipix3 Read-Out Chip basiert. Insgesamt verfügt der LAMBDA-Detektor über 12 solcher Read-Out Chips, die eine Auslesegeschwindigkeit von bis zu 2000 Bildern pro Sekunde erreichen können. Im Rahmen dieses Projektes soll nun auch die FPGA basierte FIRMWARE zur Ansteuerung des Detektors und der Datenauslese entwickelt werden. Dabei war das Ziel eine Bildwiederholungsfrequenz des LAMBDA-Detektors von bis zu 1000 Bildern pro Sekunde zu erreichen. Die Übertragung der Detektor-Daten muss sowohl über das ein Gigabit Interface als auch über ein 10-Gigabit Interface möglich sein. Dies wird es ermöglichen den Detektor mit jedem Computer zu betreiben, so dass für Experimente die keine hohen Frame-Raten benötigen, kein Bedarf an teure Server-Hardware mit 10-Gigabit Interfaces besteht. Für spezielle Experimente soll das 10-Gigabit Interface zu Verfügung stehen.

[2] Deutsches Elektronen Synchrotron

[3] ***C****onseil* ***E****uropéen pour la* ***R****echerche* ***N****ucléaire* - Europäische Organisation für Kernforschung

Zielsetzung

In Rahmen dieser Diplomarbeit soll ein Detektor Steuerungssystem in Form eines SoC[4] als FPGA Firmware Komponente entwickelt werden welche die Steuerung und Überwachung aller Einheiten und Komponenten des Detektor-Systems gewährleistet und eine Read-Out Geschwindigkeit der Medipix3 Chips von bis zu 1000 Bildern pro Sekunde ermöglicht. Da der Detektor für unterschiedliche Experimente flexibel einsetzbar sein soll, sollen zwei unterschiedliche Ethernet-Schnittstellen zur Ansteuerung des Detektors mit dem Computer und zur Datenauslese implementiert werden. Wobei das Gigabit Ethernet Interface als Einheit des SoC realisiert werden soll, welches sowohl für die Übertragung der Detektor-Daten als auch für die Übertragung der Befehle für die Steuerung des Detektors verwendet wird. Die vom PC kommenden Steuerbefehle müssen von dem SoC empfangen, Interpretiert und Ausgeführt werden.

[4] System on Chip

2 Grundlagen und Stand der Technik

Dieses Kapitel gibt einen Überblick über die Grundlagen der photonenzählenden pixelierten Röntgendetektoren und beschreibt die Grundprinzipien der Detektierung von Röntgenstrahlung. Des Weiteren wird der Medipix3 Readout-Chip vorgestellt und dessen funktionsweise erläutert. Anschließend folgt die Beschreibung der vorhandenen Detektorhardware, der Architektur des Xlinix Virtex5 FPGA's sowie des eingebetteten PPC 440 Prozessor Blocks.

Grundlagen der Röntgen Detektoren

2.1.1 Photonenzählende pixelierte hybride Halbleiterdetektoren

Wie in Abbildung 1 dargestellt, bestehen hybride Halbleiterdetektoren in der Regel aus einem Halbleitersensor und einem Readout Chip. Wobei die Verbindung zwischen dem Sensor und Readout Chip durch die so genannte Flip-Chip-Verbindungstechnologie erfolgt. Dabei werden Sensor und Elektronik mittels kleinen Lotkugeln den so genannten Bump-Bonds aus Indium oder einer Zinn-Blei-Antimon-Legierung miteinander verbunden. Durch diesen hybriden Aufbau können unterschiedliche Sensormaterialien wie Silizium, Cadmiumtellurid oder Galliumarsenid verwendet werden. Quelle [6] S.5

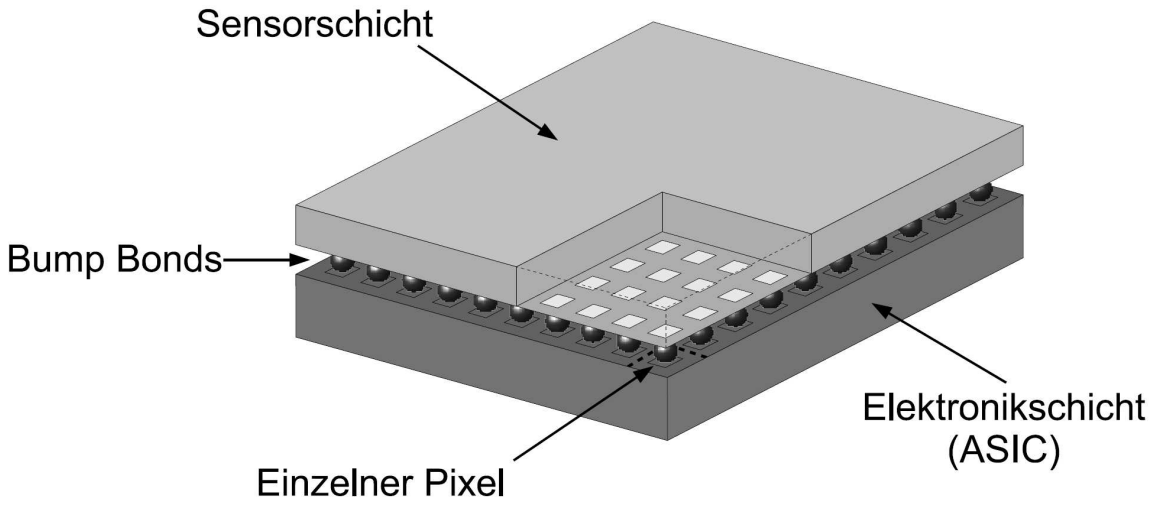


Abbildung 1: Schematischer Aufbau eines hybriden Detektors. Quelle [6] S.6

Trifft ein Rötgenphoton auf den Halbleitersensor, wird dieses durch den Photoeffekt absorbiert, wobei freie Elektronen-Lochpaare erzeugt werden. Eine am Sensor angelegte Hochspannung erzeugt ein elektrisches Feld wodurch die freien Ladungsträger zu den Pixel-

elektroden driften. Dadurch entsteht an den Elektroden ein kurzer Strompuls der über die weitere Elektronik im Chip weiterverarbeitet wird und schließlich ein digitaler Zähler inkrementiert wird. Bei dieser art von Detektoren werden also alle Photonen einzeln gezählt. Quelle [6] S.5, S.6

Medipix3 Readout Chip

Der Medipix3 Readout Chip ist ein Application Specific Integrated Circuit (ASIC) welcher am CERN[5] (Genf, Schweiz) entwickelt wurde. Der Medipix3 ist ein 2-dimensionaler pixelierter, photonenzählender Halbleiterdetektor welcher für Experimente mit Synchrotronstrahlung, sowie biomedizinische Bildgebung vorgesehen ist. Abbildung 1 stellt den Medipix3 Readout Chip dar.

Der Medipix3 Chip verfügt über eine Matrix von 256x256 Pixeln mit einer Pixelgröße von 55x55 µm² und ist 15,88 mm hoch und 14,1mm breit. Jeder Pixel verfügt über einen analogen Schaltkreis zur direkten Signalverarbeitung sowie zwei einstellbare Zähler die als 4, 6 oder 12 Bit-Zähler konfiguriert werden können. Um eine hohe Auslesegeschwindigkeit zu erreichen ist der Chip mit acht Datenausgängen ausgestattet und kann mit bis zu 200MHz getaktet werden. Bei maximaler Taktfrequenz und einer 12 Bit Zählerbreite können Frame-Raten von ca. 2034 Bilder pro Sekunde erreicht werden.

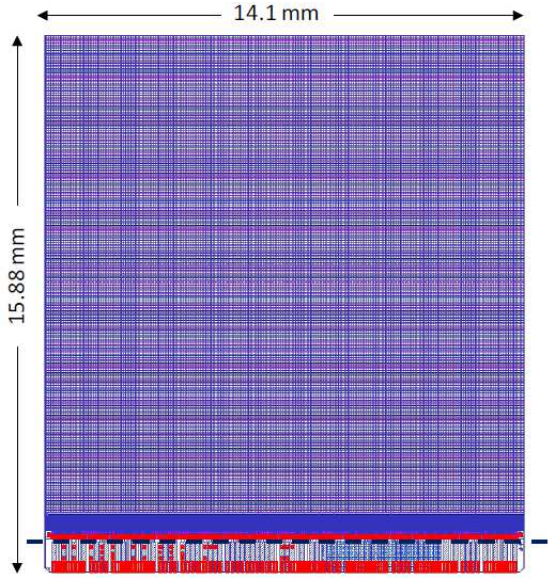


Abbildung 2: Medipix3 Read-Out Chip. Quelle [1, S.7]

2.1.2 Physische Beschreibung des Chips

Das Blockschaltbild eines Pixels ist in der Abbildung 3 dargestellt. Jeder der 65536 Pixel des Chips verfügt über einen analogen und einen digitalen Schaltkreis. Der analoger Schaltkreis beinhaltet einen Ladungsvorverstärker, auch „Ladungs-Spannungs-Wandler“ genannt und einen Testkondensator welcher zum Einkoppeln von Ladungen in den Chip unabhängig vom Sensor verwendet wird. Der Eingang des Vorverstärkers wird mit dem Pixel eines Sensors Verbunden. Als Sensor dient eine Matrix aus Photodioden die auf einem Siliziumsubstrat aufgebracht sind. Der Ausgang des Vorverstärkers ist durch einen „Shaper“ mit zwei Diskriminatoren verbunden. Der Shaper agiert als Transkonduktanzverstärker welcher die Eingangspannung in einen proportionalen Ausgangsstrom umwandelt und gleichzeitig dient er als Gauß-Filter für die Impulsformung [1]. Die beiden Diskrimi-

[5] European Organization for Nuclear Research (Conseil Européen pour la Recherche Nucléaire)

natoren werden zur Unterscheidung der Nutzsignale von anderen, kleineren Impulsen verwendet die aufgrund des Rauschens entstehen können.

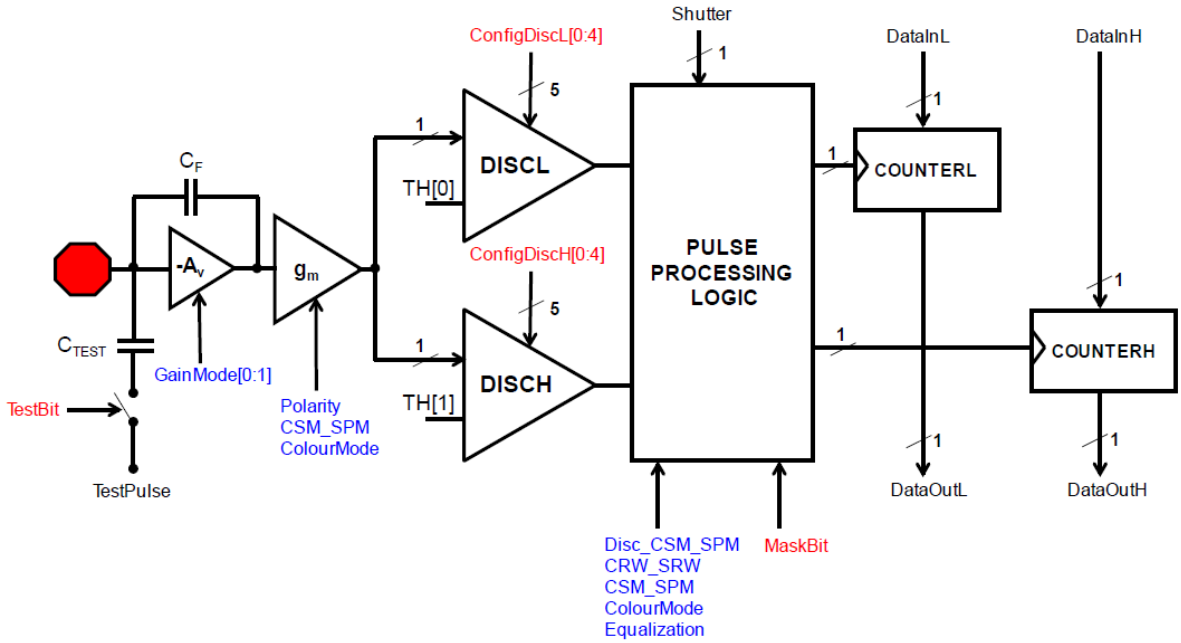


Abbildung 3: Blockschaltbild eines Pixels des Medipix3 Readout Chips. Quelle [1] S.12

Der digitale Schaltkreis des Pixels beinhaltet eine „Pulse Processing Logic“ und zwei Linear Feedback Shift Register. Jeder Pixel verfügt über zwei verschiedenen Arbeitsmodi zwischen denen durch den „Shutter“ Eingang der „Pulse Processing Logic“ umgeschaltet werden kann. Wenn das Shutter-Signal logisch „0“ ist, dann ist einen „Data Acquisition Mode“ aktiviert. In diesem Fall funktionieren die Linear Feedback Shift-Register als “pseudo-random Counter”, wobei die zwei Ausgänge der „Pulse Processing Logic“ als Taktquellen für die beiden Zähler dienen.

Sollte die Oberfläche des Sensors von einem Photon getroffen werden, entsteht einen Stromimpuls am Eingang des Vorverstärkers. Der Stromimpuls wird verstärkt und durch den Shaper geformt. Wird die Amplitude des Impulses größer als der eingestellte Schwellenwert (Threshold) am Diskriminator, so wird der Impuls zur „Processing Logic“ weitergegeben. Durch die „Pulse Processing Logic“ erfolgt eine weitere Aufbereitung des Impulses welcher letztendlich den Eingang des Zählers wirkt. Jeder von der „Pulse Processing Logic“ kommende Impuls inkrementiert dadurch den Zähler um „eins“. So ist es zum Beispiel möglich, durch die Anzahl der gezählten Photonen pro Zeiteinheit die Intensität der Röntgen Strahlung zu ermitteln.

Wenn das Shutter-Signals logisch ‚1’ ist, wird der „Pixel Readout Mode“ initialisiert. In diesem Modus werden die Shift Register jedes Pixels, mit den vor- und nachfolgenden Shift Regestern der benachbarten Pixel verbunden. Durch diese Anordnung der Register werden 256 Spalten der Matrix mit je 3072-bit, entsprechend der Abbildung 4 geformt. Die Kommunikation der Pixelmatrix mit I/O-Logik erfolgt mittels des 256-Bit „Fast Shift Register“.

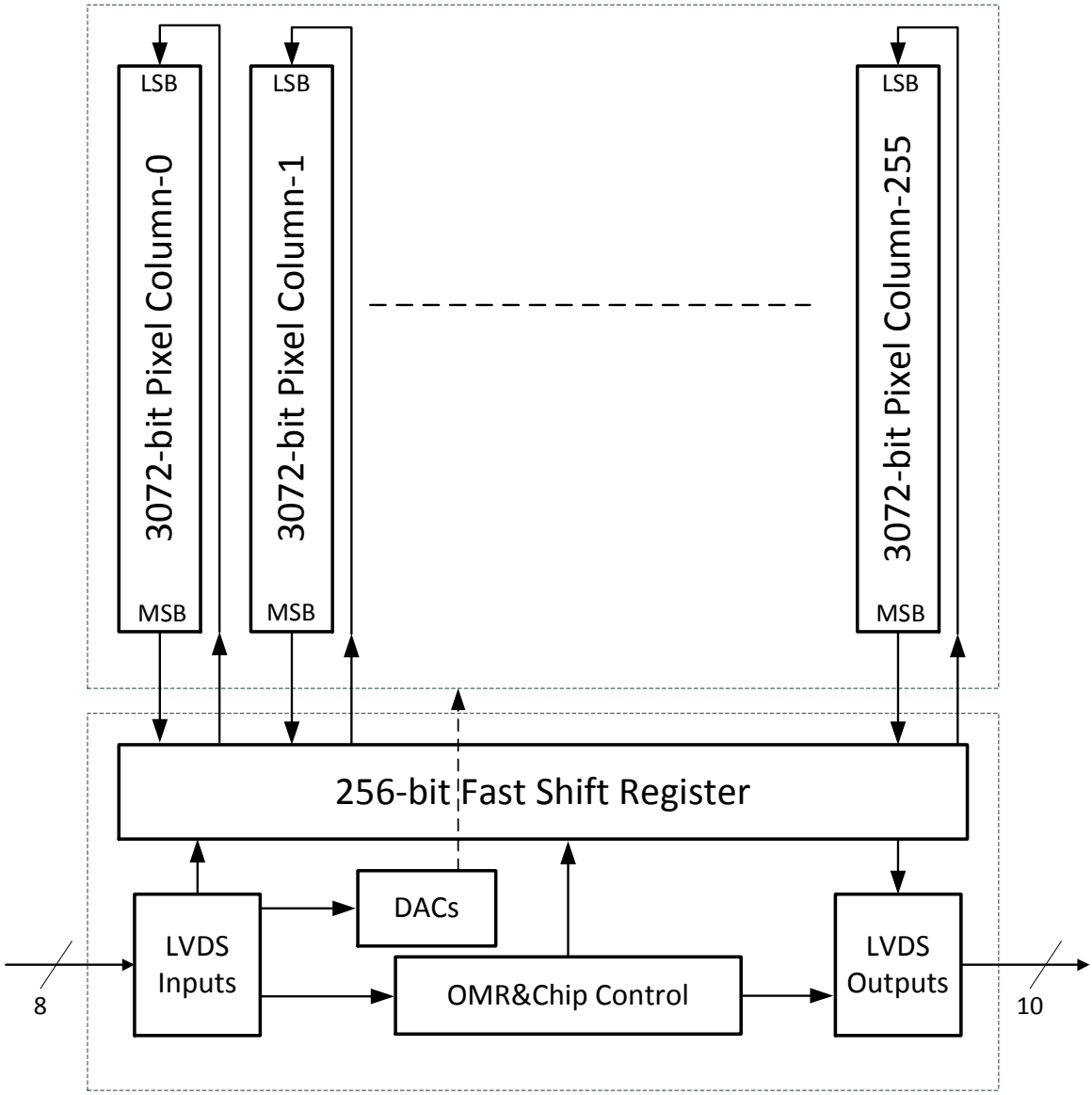


Abbildung 4: Matrix-Organisation des Medipix3 Chips

Der Medipix3Rx Chip verfügt über 27 Interne Digital-analog Wandler, die zum Einstellen von Bias- und Referenzwerte der analogen Schaltkreise der Pixel verwendet werden. Die Initialisierung der DACs erfolgt durch den digitalen seriellen Dateneingang des Chips. Die Ansteuerung des Chips erfolgt mittels Konfiguration des „Operation Mode Register“ sowie der analoge Eingänge des Chips.

Des Weiteren verfügt der Medipix3 Chip über zwei unterschiedliche Readout Modi: Sequentiellen Read-Write Modus und Kontinuierlichen Read-Write Modus. In dem Sequentiellen Read-Write Modus sind die beiden Zähler gleichzeitig im Zählbetrieb (Data Acquisition Mode) und können nacheinander ausgelesen werden. In dem Kontinuierlichen Read-Write Modus ist der gleichzeitige Zähl- und Readout Betrieb möglich. Während einer der beiden Zähler die Photonen zählt, kann anderen Zähler gelesen werden und umgekehrt. In diesem Modus ist ein totzeitfreier Betrieb möglich.

2.1.3 Operation Mode Register (OMR)

Das Operation Mode Register abgekürzt „OMR-Register“ ist 48-Bit Breit. Durch das Konfigurieren des OMR-Registers wird die Funktionalität des Chips programmiert. Das OMR-Register muss vor der Ausführung einer neuen Operation durch den Dateneingang des Chips seriell geladen werden. Nach dem Reset des Chips werden alle Bits des OMR-Registers gelöscht, mit Ausnahme des „Polarity“-bits, dieser wird gesetzt. Der Inhalt des OMR-Registers kann nur durch Reset oder serielles Laden des Registers verändert werden.

In der Tabelle 1, die dem Medipix3 Manual entnommen wurde sind die Funktionen der Bits des OMR-Registers zusammengefasst.

Name	Bits	Default	Description
M[0:2]	0:2	000	Operation mode selection: 000: Read CounterL 001: Read CounterH 010: Load CounterL 011: Load CounterH 100: Load DACs 101: Load CTPR 110: Read DACs 111: Read OMR and ChipID
CRW_SRW	3	0	0:Sequential Read Write 1:Continuous Read Write
Polatiry	4	1	0: Electron collection mode 1: Holes collection mode
PS[0:1]	5:6	00	00: DataOut[0] 10: DataOut[0..1] 01: DataOut[0..3] 11: DataOut[0..7]
Disc_CSM_SPW	7	0	Selects which Discriminator output is used during the 24-bit mode, CRW or Equalization operation modes 0: Selects DiscL 1: Selects DiscH
Enable_TP	8	0	1: Enable internal Test pulse 0: Disable external Test pulse
CountL[0:1]	9:10	00	Selects the counter depth: 00: 2x 1-bit 10: 2x 6-bit 01: 2x 12-bit 11: 1x24 bit
ColumnBlock[0:2]	11:13	000	Selects the matrix columns to be readout if ColumnBlockSel is high. See 7.6.2.1 000 100 010 110 001 101 011 111
ColumnBlockSel	14	0	0: All columns are read out 1: Matrix columns selected by ColumnBlock[0:2] are read out

RowBlock[0:2]	15:17	000	Selects the matrix rows to be readout if RowBlock-Sel is high. See 7.6.2.1. 000: Row [0] 100: Row [0:1] 010: Row [0:3] 110: Row [0:7] 001: Row [0:15] 101: Row [0:31] 011: Row [0:63] 111: Row [0:127]
RowBlockSel	18	0	0: All rows are read out 1: Matrix rows selected by RowBlock[0:2] are read out
Equalization	19	0	Selects if the chip is in Threshold Equalization Mode (see chapter 8.4) 0: Equalization OFF 1: Equalization ON
ColorMode	20	0	0: Fine Pitch Mode (55µm x 55µm) 1: Spectroscopic Mode (110µm x 110µm)
CSM_SPM	21	0	0: Single Pixel Mode 1: Charge Summing Mode
InfoHeader	22	0	0: No Info header added 1: Info header (OMR bits + chip ID) added before the DataOut stream in Data_Out[0]
FuseSel[0:4]	23:27	000000	Selects Fuse to be burned
FusePulseWidth[0:6]	28:34	00000000	Set the programming pulse width to burn the selected e-fuse 0000_000: No pulse 1000_000: 1024 clocks width 1111_111: 523264 clocks width
GainMode[0:1]	35:36	00	00: Super-High Gain Mode (SHGM) 10: High Gain Mode (HGM) 01: Low Gain Mode (LGM) 11: Super-Low Gain Mode (SLGM)
SenseDAC[0:4]	37:41	00000	Selects DAC to be sensed through DACOUT pin
ExtDAC[0:4]	42:46	00000	Selects DAC to be externally imposed through EXT-DAC pin
ExtBGSel	47	0	0: Internal Band-Gap is used 1: External Band-Gap used through EXTBG pin

Tabelle 1: Operation Mode Register. Nach Quelle [1] S.35

2.1.4 Chip-Interface

Der Medipix3 Chip verfügt insgesamt über zwei Single-Ended sowie 8 LVDS[6]-Eingänge. Für die Konfiguration der Chip-Internen Register ist ein seriellen Dateneingang vorgesehen. Der Chip verfügt über keine eigene Taktquelle und muss mit einem externen Takt versorgt werden, hierfür ist einen Clock-Eingang vorhanden. Für die Ansteuerung der Pixel-Matrix sowie Chip-interner Logik sind weitere 6 LVDS-Eingänge vorgesehen, deren Funktionsweise nachfolgend erklärt wird.

[6] Low Voltage Differential Signaling

Wie oben bereits erwähnt, verfügt der Madipix3 Chip über 27 Digital-Analog Konverter die zum Einstellen der Bias- und Referenzwerte verwendet werden. Die Eigenschaften der Halbleiter des Chips können durch starke ionisierende Strahlung während des Experimentes verändert werden wodurch die Digital-Analog Wandler nicht mehr korrekt funktionieren können. Deshalb verfügt der Chip über zwei analoge „Single-Ended“ Eingänge „ExtBG_in“ und „ExtDAC_in“ die erlauben einen der Internen A/D-Wandler durch externe Spannungsquelle zu ersetzen. Zum Überwachen der Referenzwerte die durch die D/A Wandler erzeugt werden ist ein „Single-Ended“ Ausgang „DAC_Out“ vorgesehen, wobei jeder D/A Wandler an diesen Ausgang durch das Konfigurieren des Operation Mode Registers geschaltet werden kann.

Zum Lesen der Pixel-Matrix sowie der internen Chip-Register ist der Medipix3 Chip mit 8 LVDS-Datenausgängen ausgestattet. Die Pixel-Matrix kann entweder durch acht, vier, zwei oder einen Datenausgang gelesen werden. Dies wird durch das Konfigurieren der OMR[7]-Register festgelegt. Das Lesen der Chip-Register erfolgt nur durch einen Daten-Ausgang. Für die zeitliche Synchronisation während der Datenübertragung und Quittierung der ausgeführten Chip-Operationen sind „Enable-Out“ und „Clock-Out“ Ausgänge vorgesehen.

Der Medipix3 Chip ist in der Abbildung 5 schematisch dargestellt. In der Tabelle 2 sind die Beschreibungen der Chip Ein- und Ausgänge zusammengefasst.

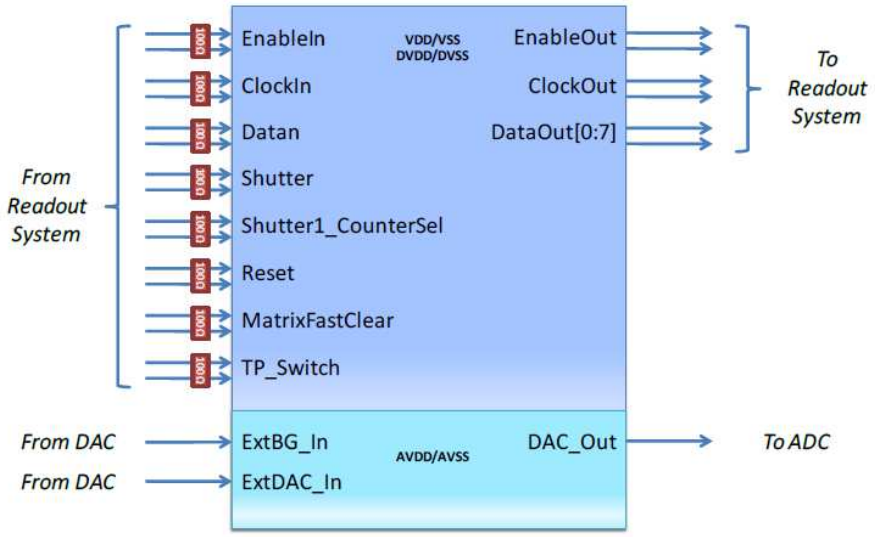


Abbildung 5: Medipix3 Interface. Quelle [1] S.51

Bezeichnung	Signal-Typ	In/Out	Beschreibung
EnableIn	LVDS	In	Aktiviert I/O-Logik
ClockIn	LVDS	In	Takteingang
DataIn	LVDS	In	Dateneingang

[7] Operation Mode Register.

Shutter	LVDS	In	Steuert Acquisition-Zeit, schaltet die „Acquisition“ und „Readout“ Modi um.
Shutter1_CounterSel	LVDS	In	Schaltet zwischen Counter0 und Counter1 um.
Reset	LVDS	In	Reset von DACs und interner Logik
MatrixFastClear	LVDS	In	Asynchrones Pixelmatrix Reset
TP_Switch	LVDS	In	Test-Pulse Eingang
ExtBG_Int	analog	In	Externer Referenzspannungseingang für D/A Wandler
ExtDAC_In	analog	In	Analoger Testeingang der D/A Wandler
EnableOut	LVDS	Out	Kennzeichnet durch die fallende Flanke Ende der Datenübertragung
ClockOut	LVDS	Out	Taktausgang, hat Phasenverschiebung von 180° sowie 4ns Delay relativ zum Eingangstakt
DataOut[0..7]	LVDS	Out	8 Daten-Ausgänge, Anzahl der verwendete Daten-ausgänge ist durch OMR-Register konfigurierbar.

Tabelle 2: Beschreibung der Chip Anschlüsse

2.1.5 Ansteuerung des Chips

2.1.5.1 Chip Reset

Der Reset-Vorgang des Medipix3 Chips wird durch das Schalten des Reset-Inputs auf Logische „0“ eingeleitet. Wenn der Reset-Input logische „0“ aufweist werden zum Zeitpunkt der nächste steigende Flanke des Eingangstakte alle interne Zustandsautomaten in ihren Anfangszustand gebracht, alle interne DAC’s auf ihren jeweiligen Vorgabewert zurückgesetzt und der Inhalt des Operation Mode Registers wird gelöscht. Diese Operation hat die höchste Priorität.

2.1.5.2 Chips Operationen

Für die Konfiguration und Ansteuerung des Medipix3 Chips stehen drei Modi zur Verfügung:

1) **Laden des Operation Mode Registers (OMR):** Durch diese Operation wird OMR bitseriell geladen, wodurch eine Konfiguration des Chips stattfindet.

2) **Start einer Operation:** Durch dieses Vorgang wird die zuvor im OMR-Register vorprogrammierte Operation ausgeführt

3) **Brennen der E-Fuses:** Standalone Operation, wird zum brennen der „E-Fuses“ zur Identifikation des Chips verwendet.

Die drei oben genannten Operationsmodi werden durch ein 4-Bit Header-Word eingeleitet, welches bei jeder Kommunikation mit dem Chip verwendet werden muss.

Die drei Header-Words sind wie folgt definiert:

- 0011LSB: Laden des Operation-Mode-Registers
- 0101LSB: Ausführen einer Operation
- 0010LSB: Brennen der E-Fuses

Der Medipix3 Chip verfügt über einen Zustandsautomat welcher die Eingangsdaten des Chips kontrolliert und auf die oben beschriebenen Bitmuster wartet, wenn der Chip aktiv ist. (Enable-Input ist „low“)

Beim Konfigurieren, Ansteuern und Auslesen des Medipix3 Chips muss beachtet werden, dass die „Reset“, „DataIn“, „EnableIn“, „MatrixFastReset“, „Shutter“ and „CounterSelCRW“ Eingänge bei einer steigenden Flanke des Eingangstaktes abgetastet werden. Die Datenausgänge (DataOut[0:7]) und „EnableOut“ Ausgang hingegen werden an einer fallenden Flanke des Taktes am „ClockOut“ Ausgang gesetzt, wobei der Ausgangstakt des Chips eine Phasenverschiebung von 180° sowie eine 4 ns Zeitverzögerung relativ zum Eingangstakt hat. [1] S.32

2.1.5.3 Laden des Operation Mode Register

Abbildung 6 zeigt die Datensequenz welche zum Laden des OMR-Registers verwendet wird. Diese Datensequenz ist 80 Bit lang und besteht aus einem 16 Bit-Header für die Pre-Synchronisation, 48 Bit OMR-Daten und 16 Post-Synchronisationsbits.

Als Header wird in diesem Fall eine (0x30, 0x00) Bitfolge verwendet, wodurch der Ladevorgang des Operation Mode Registers eingeleitet wird. Danach werden die nachfolgenden 48 Bit in das OMR des Chips bitseriell geschrieben. Schließlich wird diese Operation durch eine (0x00, 0x00) Bitfolge beendet.

Damit der Chip die Kommandos annehmen kann, muss er vor Beginn der Datenübertragung aktiviert werden, dies geschieht durch das Schalten des „EnableIn“ Eingangs auf Pegel „Low“.

Die korrekte Übertragung der Datensequenz zum Chip wird von diesem mit einer bitseriellen Ausgabe der Bitfolge „0011“ am „DataOut[0]“ Ausgang bestätigt. Wird die jeweilige Operation erfolgreich vom Chip ausgeführt wird dies durch einen Pegelwechsel von „Hi“ auf „Low“ am „EnableOut“ Ausgang vom Chip quittiert.

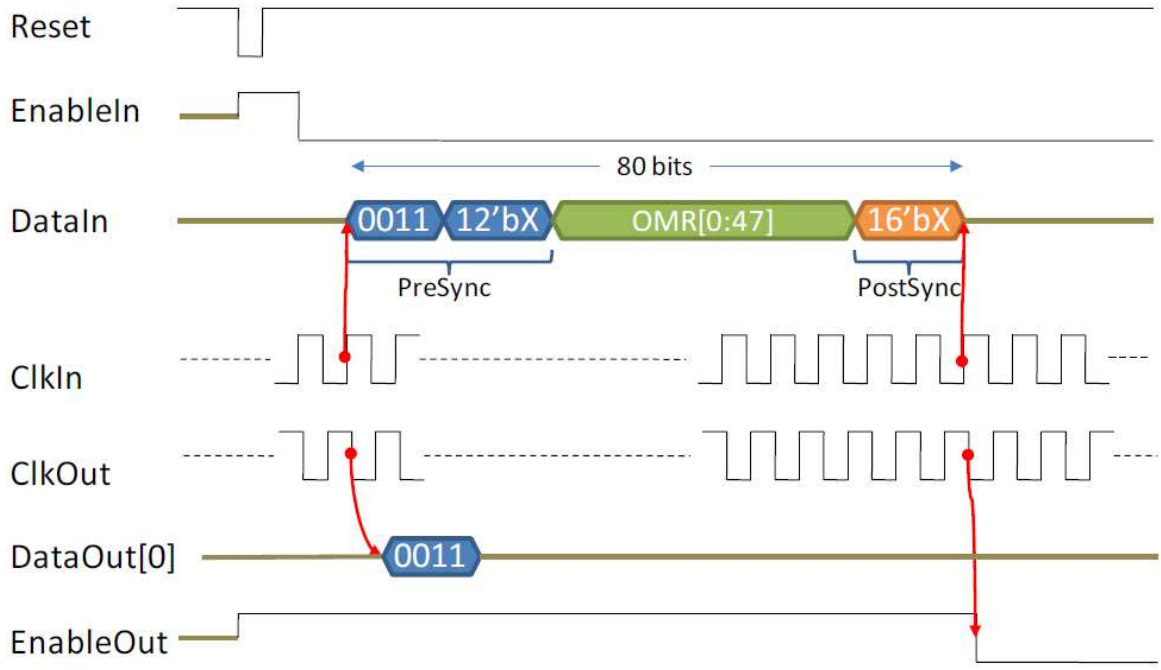


Abbildung 6: Laden des OMR-Registers. Quelle [1] S.35

2.1.5.4 Peripherie Operationen

Das Laden und Lesen der DACs des Medipix3-Chips wird als Peripherie Operation bezeichnet. Das Ausführen dieser Operation erfolgt durch das Laden der von festgelegten Befehlssequenzen. Alle diese Sequenzen sind 288 Bit Lang und sind ähnlich aufgebaut: 16 Bit Pre-Synchronisation, 256 Input oder Output Bits und 16 Post-Synchronisationsbits. Zum Festlegen der Peripherie Operationen werden die so genannten drei „M“-Bits (M0, M1 and M2) des OMR-Registers verwendet, die wie folgt konfiguriert werden können:

- Laden der DACs (M012=100)
- Lesen der DACs (M012=110)

Um eine der oben genannten Operationen auszuführen muss das OMR-Register des Chips vorher geladen werden.

In Abbildung 7 ist der Ladevorgang der 25 DAC-Register des Chips dargestellt. Die OMR- sowie DAC-Daten werden durch den „DataIn“ Eingang des Chips bitseriell geladen.

Der Lesevorgang der DAC-Register stellt Abbildung 8 dar. Zum Lesen der DACs wird erst das OMR-Register geladen. Anschließend wird durch die Übertragung der Execution-Sequenz (0xA0, 0x00) der Inhalt der DAC-Register durch den „DataOut[0]“ Ausgang bitseriell ausgegeben.

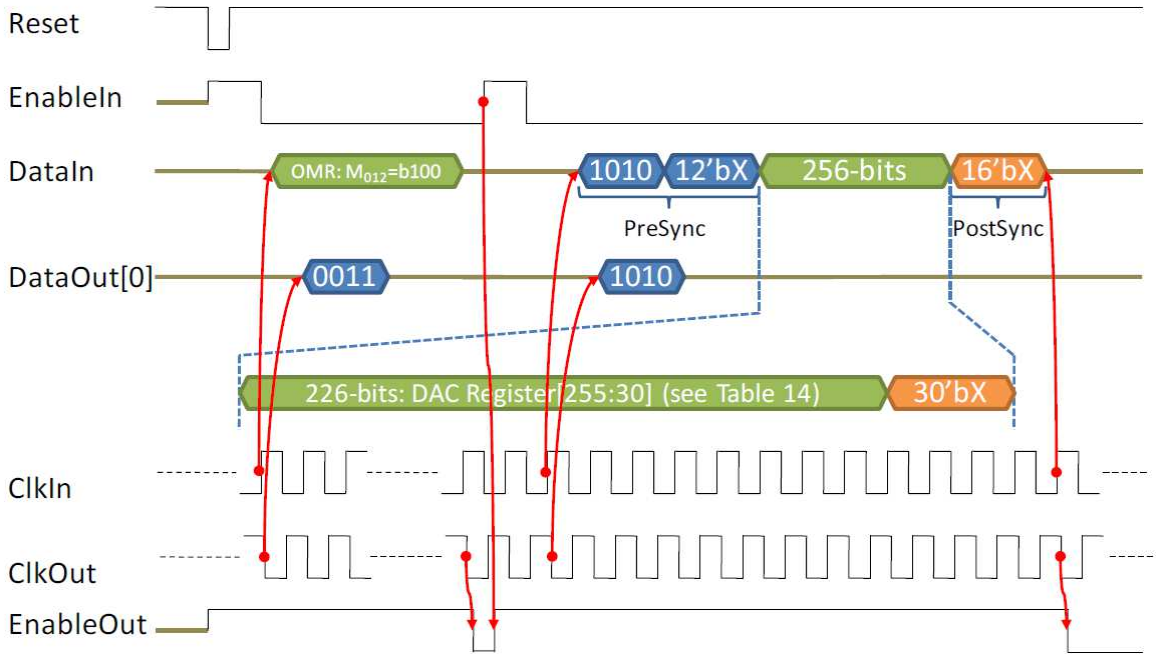


Abbildung 7: Laden der DAC-Register. Quelle [1] S.36

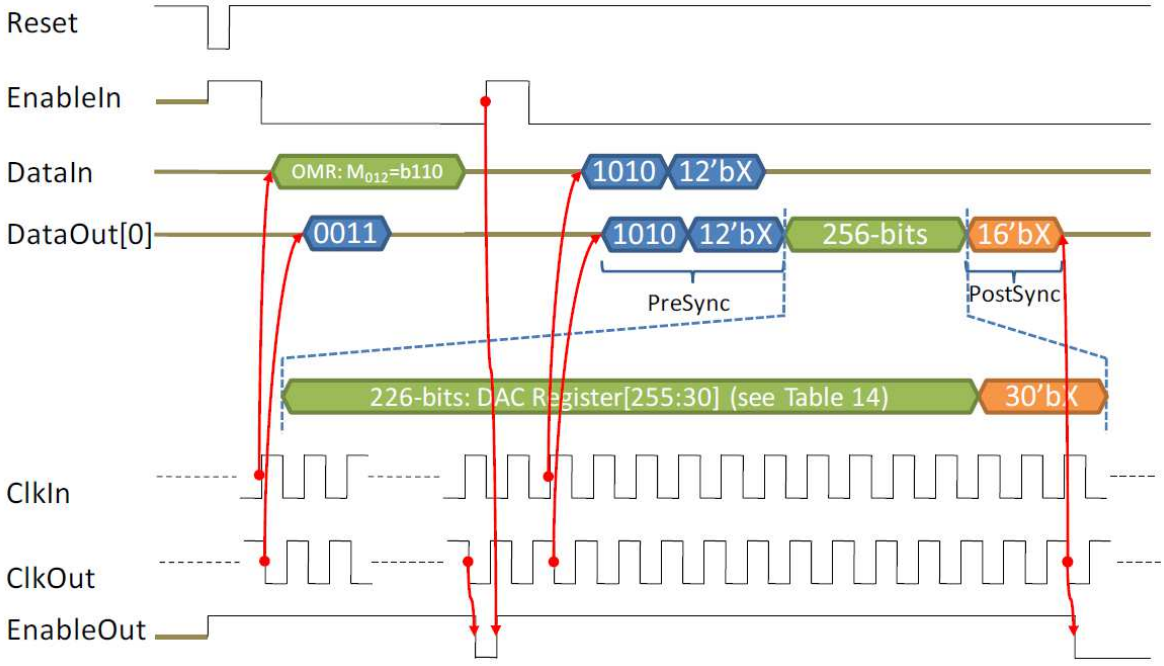


Abbildung 8: Lesen der DAC-Registers. Quelle [1] S.37

2.1.5.5 Pixel-Matrix Operationen

Das Lesen der Counter-Matrizen des Medipix3-Chips wird als Matrix-Operation bezeichnet, Zur Auswahl der jeweiligen Matrix-Operation werden ebenfalls die „M"-Bits des OMR-

Registers verwendet. Um den Lesevorgang der Pixel-Matrix einzuleiten werden die „M“-Bits des OMR-Registers wie folgt konfiguriert:

- Lesen der Pixel-Matrix, Counter 0 (M012=000)
- Lesen der Pixel-Matrix, Counter 1 (M012=001)

Der Lesevorgang der Pixel-Matrix durch zwei Datenausgänge „DataOut[0]“ und „DataOut[1]“ ist in der Abbildung 9 dargestellt. Alternativ kann die Pixel-Matrix durch einen, vier oder acht Datenausgänge gelesen werden. Zum Definieren der Datenausgänge zum Lesen der Matrix werden die „PS“-Bits des OMR-Registers verwendet.

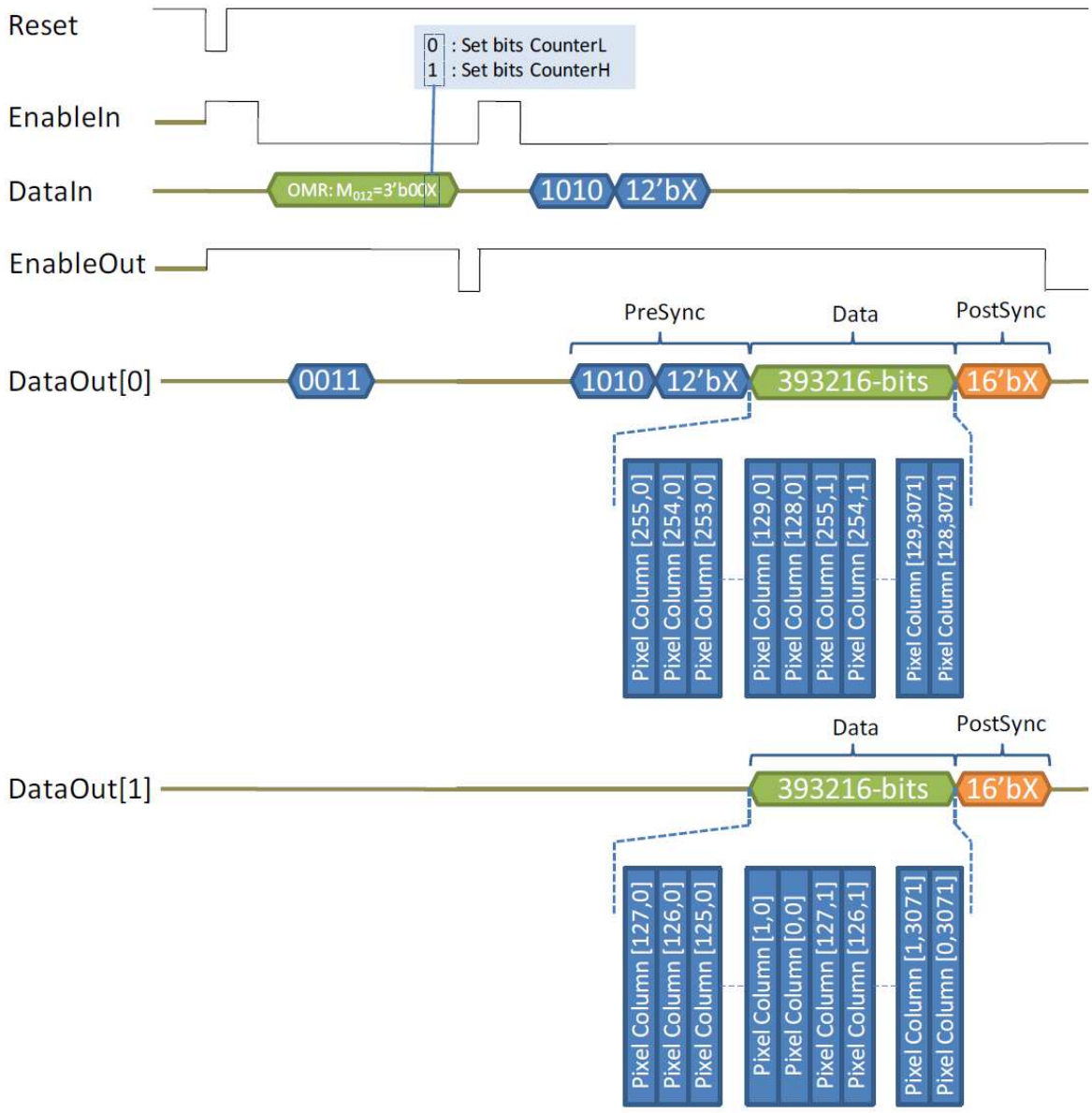


Abbildung 9: Das Lesen der Pixel-Matrix. Quelle [1] S.42

2.1.5.6 Acquisition Modi

Zum Auslesen der Pixel-Matrix des Medipix3 Chips sind zwei verschiedenen Modi implementiert. Zum Festlegen eines Readout Modus werden die „CRW_SRW“-Bit des OMR-Registers sowie die Shutter and CounterSelCRW Eingänge verwendet.

- **Sequential Acquisition Modus:** Der Lesevorgang im sequential acquisition Modus ist in der Abbildung 10 dargestellt. In diesem Modus erfolgt das Zählen der Photonen und Auslesen der Pixel-Matrix sequentiell, wobei der Zählbetrieb durch das aktivieren des Shutter Signals initiiert wird. Das Festlegen des Zählers der gelesen werden soll, erfolgt durch die Konfiguration des OMR-Registers.
- **Continuous Read Write Acquisition Modus:** Der Lesevorgang unter Verwendung des continuous read write Modus stellt die Abbildung 11 dar. In diesem Modus werden die beiden Zähler der Pixels verwendet. Während einer der beiden Zähler die Photonen zählt kann der zweite Zähler gelesen werden und umgekehrt. Das Umschalten der Zähler erfolgt in diesem Modus durch den „Shutter1_CounterSel“ Eingang.

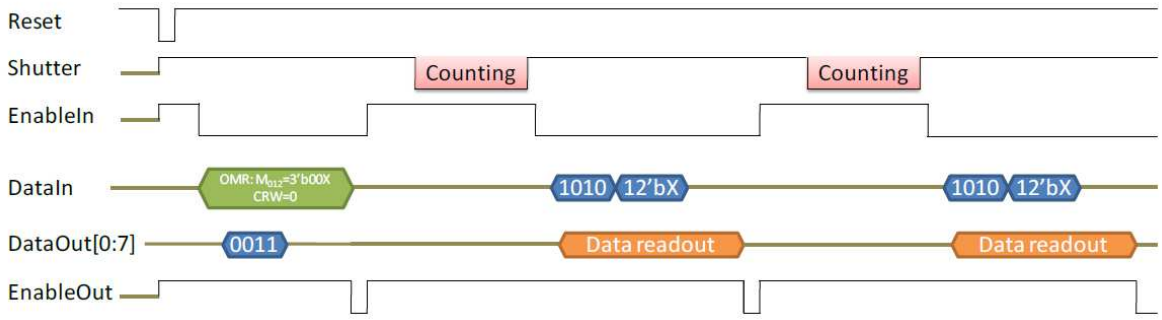


Abbildung 10: Das Lesen der Pixel-Matrix im Sequential Acquisition Modus. Quelle [1] S.48

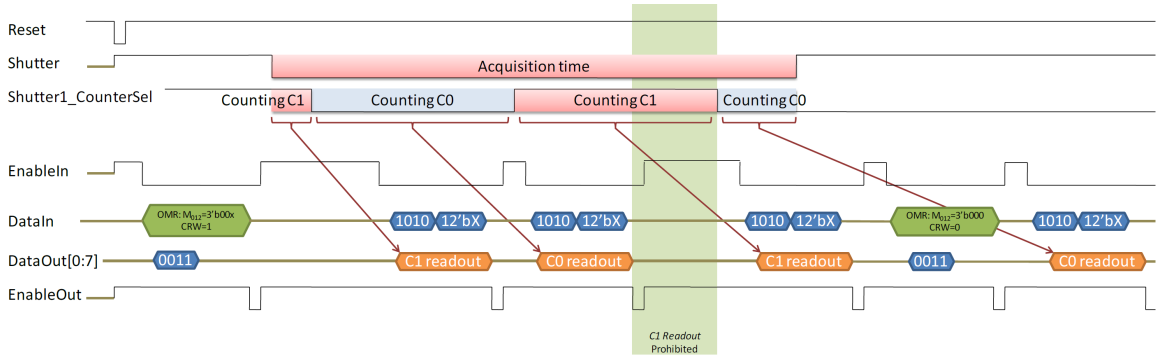


Abbildung 11: Das Lesen der Pixel-Matrix im Continuous Read Write Acquisition Modus. Quelle [1] S.49

2.1.6 Architektur des Xilinx Virtex-5 FPGAs

Ein Field Programmable Gate Array (FPGA) ist ein Integrierter Schaltkreis, in dem verschiede logische Schaltungen Programmiert werden können. Anders als bei Mikroprozessoren bezieht sich der Begriff „Programm“ nicht auf die Vorgabe der zeitlichen Abläufe im Baustein, sondern auf die Definition von dessen Funktionsstruktur. Durch die Programmierung von Strukturvorschriften wird zunächst die grundlegende Funktionsweise einzelner konfigurierbarer Logik-Blöcke im FPGA und deren Verschaltung untereinander festgelegt. Man spricht daher auch von der Konfiguration eines FPGAs.[8]

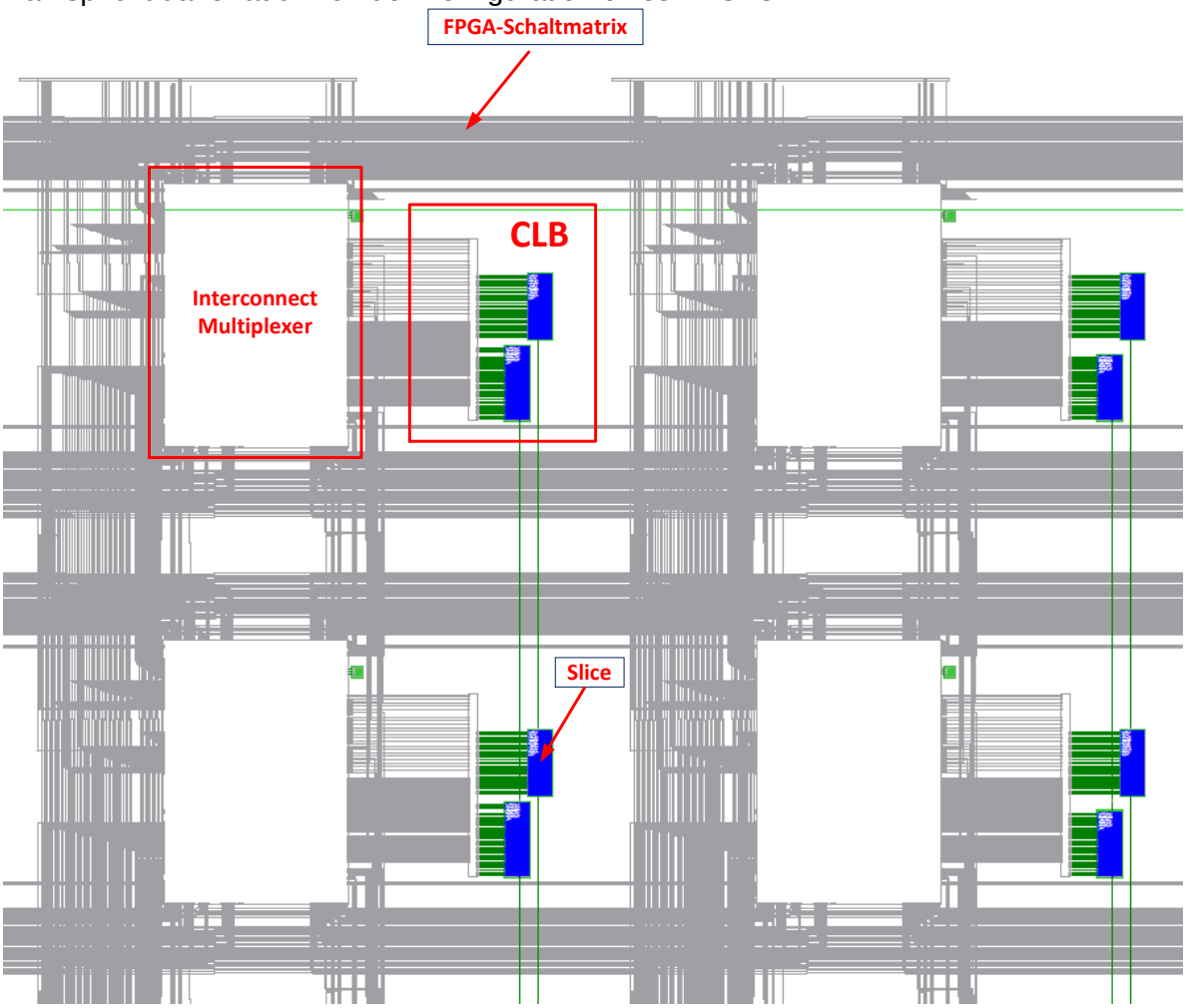


Abbildung 12: Architektur des Virtex5 FPGAs

Die wesentliche Grundstruktur eines FPGA ist ein Feld, bestehend aus Configurable Logic Blocks (CLBs). In den FPGAs der Vitrex5 Familie verfügen die CLBs jeweils über zwei Slices, welche wiederum die sogenannten LUT’s[9] für kombinatorische Funktionen sowie ver-

[8] http://de.wikipedia.org/wiki/Field_Programmable_Gate_Array

[9] Lookup Table

schiedene Flip-Flops, Register und sogar verteilte RAM-Blocke für sequenzielle Strukturen beinhalten. Die Abbildung 12, die dem Xilinx FPGA-Editor entnommen wurde zeigt der Aufbau dieser Architektur eines Xilinx „XC5VFX70T“ FPGAs. Innerhalb jedes CLB sind Verdrahtungsressourcen in Form eines Multiplexers vorhanden, die ein lokales Verschalten der Logikzellen ermöglichen. Die Verschaltung der CLBs erfolgt durch eine globale Schaltmatrix, die im „XC5VFX70T“ FPGA 160 Zeilen und 38 Spalten bildet. Die Konfiguration der Schaltmatrix und der lokalen Multiplexer wird durch das FPGA-Programm festgelegt.

Des Weiteren werden in den FPGAs auch hoch dedizierte RAM-Blöcke, Digitale Clock Manager, Hardware-Multiplizierter und weitere komplexe Komponenten integriert. Als Beispiel hierfür kann der im Xilinx „XC5VFX70T“ FPGA eingebetteten PowerPC440 Prozessorblock und eine als statische Hardware integrierte Ethernet-Tree-Mode-MAC[10] genannt werden.

2.1.7 Eingebetteter Prozessor Block

Abbildung 13 stellt ein Blockschaltbild eines im Virtex5 FPGA eingebetteten Prozessor Blocks dar. Dabei handelt es sich um einen Hard-Core der im Virtex5 FPGA statisch integriert ist. Der eingebeteter Prozessor-Block verfügt über folgende Hauptkomponenten: der PowerPC 440 Prozessor, ein „Crossbar“ und seine Interfaces, ein Auxiliary Processing Unit (APU) Controller sowie ein Clock und Reset Modul.

2.1.7.1 Komponenten-Übersicht

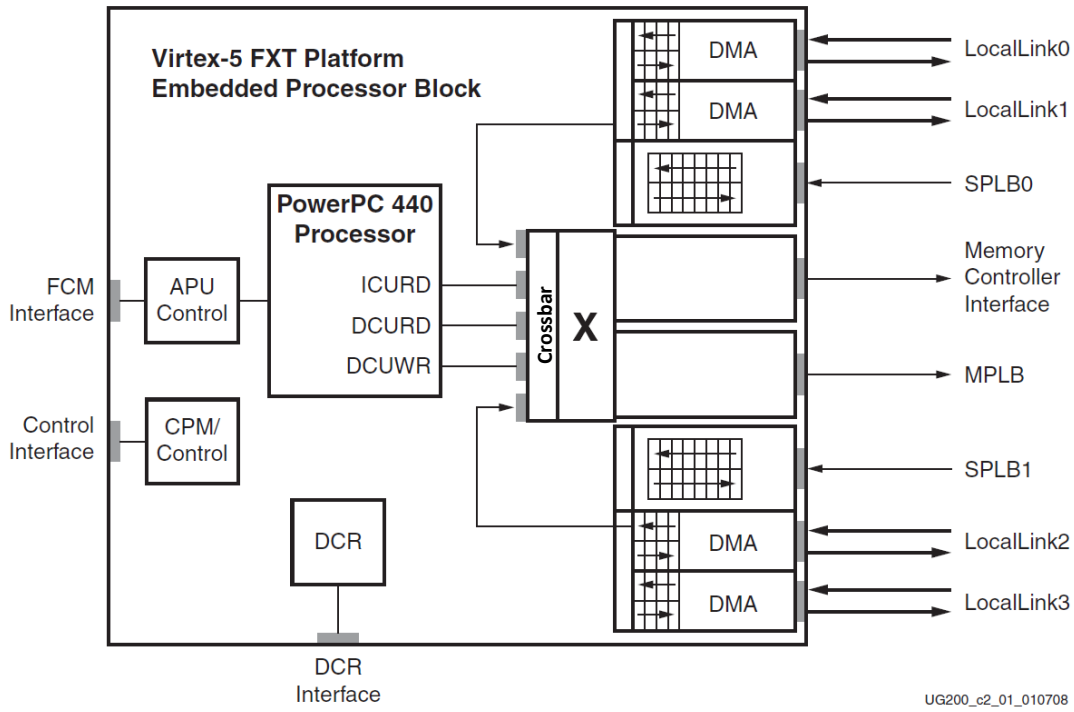


Abbildung 13: Embedded Processor Block. Quelle [3] S.29

[10] Ethernet Tree-Mode Media Access Control

2.1.7.2 PowerPC 440 Prozessor

Der PowerPC 440 ist eine 32-Bit Reduced Instruktion Set Computer (RISC) CPU[11], dabei handelt es sich um eine erweiterte „IBM Book-E“ Architektur. [3] S.17 Auf die Architektur dieses Prozessors wird im Kapitel 4.2.1 näher eingegangen. Der PowerPC 440 Prozessor verfügt über drei PLB[12] Schnittstellen die zum Lesen der Instruktion sowie zum Lesen und Schreiben der Daten verwendet werden. Durch die drei oben genannten PLB Schnittstellen erfolgen die Schreib- und Lesezugriffe auf einen in der Regel externen Speicher, welcher als Arbeitspeicher für den Prozessor agiert. Die Anbindung der „Memory mapped“ Peripherie Komponenten des Systems erfolgt etweder über das Master Processor Local Bus Interface (MPLB) oder das Slave Processor Local Bus Interface (SPLB). Des Weiteren besteht auch die Möglichkeit die Peripherie-Komponenten mittels von LocalLinks an den Prozessor Block anzuschließen.

2.1.7.3 Crossbar und seine Schnittstellen

Beim sogenannten „Crossbar“ handelt es sich um einen schnellen Multiplexer, welcher es dem PowerPC440 Prozessor erlaubt mit seinen drei PLB Schnittstellen sowie den Peripherie Komponenten, die durch MPLB, SPLB oder Local Links mit dem Prozessor Block verbunden sind, auf einen externen Speicher zuzugreifen. Dieser wird durch den Memory Controller und das Memory Controller Interface (MCI) ans System angeschlossen. Der Datentransfer durch den Crossbar erfolgt Paketorientiert im Zeitmultiplexverfahren. Wie in der Abbildung 13 gezeigt ist, verfügt der Crossbar über folgende Schnittstellen:

- Fünf PLB Slave Interfaces:
  - Drei PLB Interfaces zum Anschließen des PowerPC440 Prozessors
  - Zwei weitere PLB Interfaces erlauben einen Zugriff den Periphere-Komponenten auf das Memory Controller Interface, zum Beispiel einem Subprozessor.
- Vier Voll-Duplex Local Link Kanäle mit einem DMA-Controller welcher der Peripherie ebenfalls den Zugriff auf MCI und somit auf externen Speicher des Systems erlaubt.
- Einen High-Speed Memory Controller Interface (MCI) zum Anschließen eines Memory Conrollers. Dieses erlaubt den Betrieb mit verschieden externen Speichern wie zum Beispiel DDR[13]- und DDR2-SDRAMs sowie verschiedenen SO-DIMMs[14].
- Ein PLB Master Interface, erlaubt es dem PowerPC440 Prozessor auf Peripherie Komponenten in der FPGA Logik zuzugreifen.

[11] Central Processing Unit

[12] Processor Local Bus

[13] Double Data Rate-Zweiphasige Datenübertragung

[14] Small Outline Dual Inline Memory Modul

Weitere Details des Crossbars und seiner Schnittstellen können dem Kapitel 3 der Quelle [3] entnommen werden.

2.1.7.4 DMA – Kontroller

Der Direct Memory Access (DMA) Controller des eingebetteten Prozessor Blocks verfügt über vier voneinander unabhängigen DMA-Engines. Ein Local Link wird dabei als Schnittstelle zwischen den DMA-Engines und den Peripheriekomponenten des Systems verwendet. Mittels des Crossbars erlaubt der DMA Controller den direkten Datentransfer zwischen der Peripherie und dem Memory Controller ohne Beteiligung des PowerPC440 Prozessors. Für die Initialisierung und Überwachung des DMA Controllers stehen 17 Device Controll Registers (DCR) zu Verfügung die vom PowerPC440 Prozessor aus adressiert werden können.
Weitere Information zum DMA Controller kann dem Kapitel 13 der Quelle [3] entnommen werden.

Vorstellung des vorhandenen Detektor-Hardware

In der Abbildung 14 ist ein Prototyp des LAMBDA-Moduls und seine Readout Elektronik dargestellt. Die wichtigsten Einheiten des Detektors sind im Blockschaltbild der Abbildung 15 dargestellt. Die gesamte Hardware des Detektors beinhaltet folgende Einheiten: der Detektorkopf mit dem LAMBDA[15]-Modul aus Siliziumsensor und 2x6 Medipix3 Chips, ein „Power and Signal Distribution Board“ sowie ein FPGA-Basierten Readout Board.

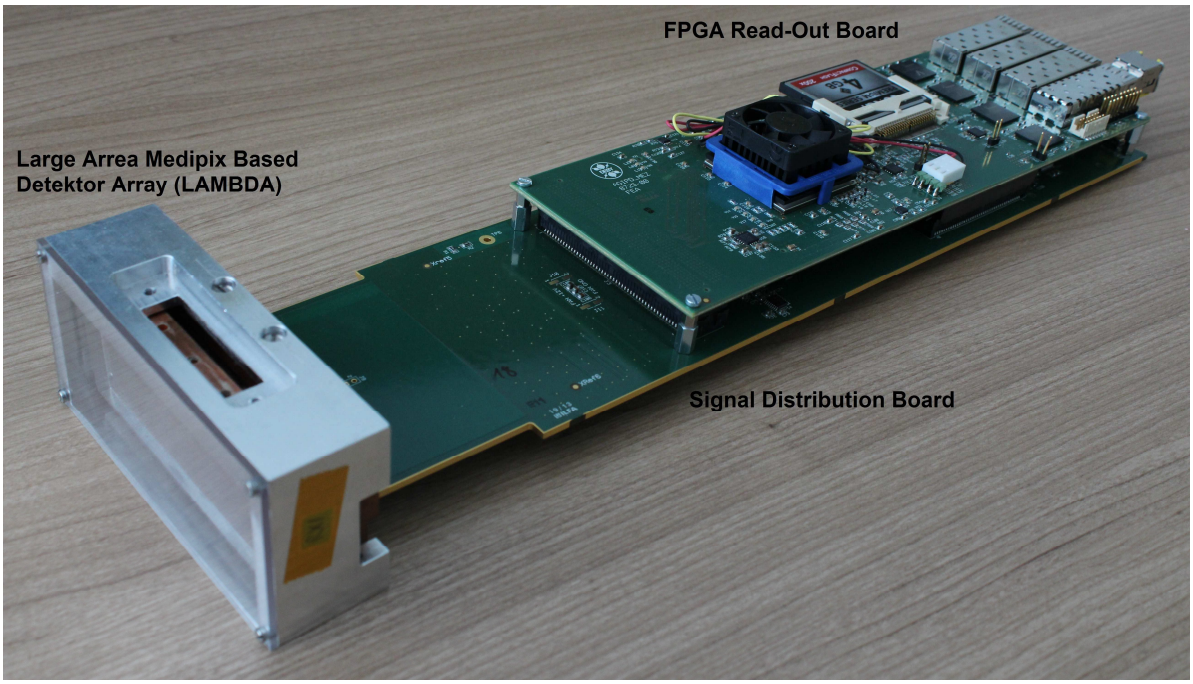


Abbildung 14: Prototyp eines LAMBDA-Moduls und deren Readout Elektronik

[15] Large Arrea Medipix Based Detektor Array

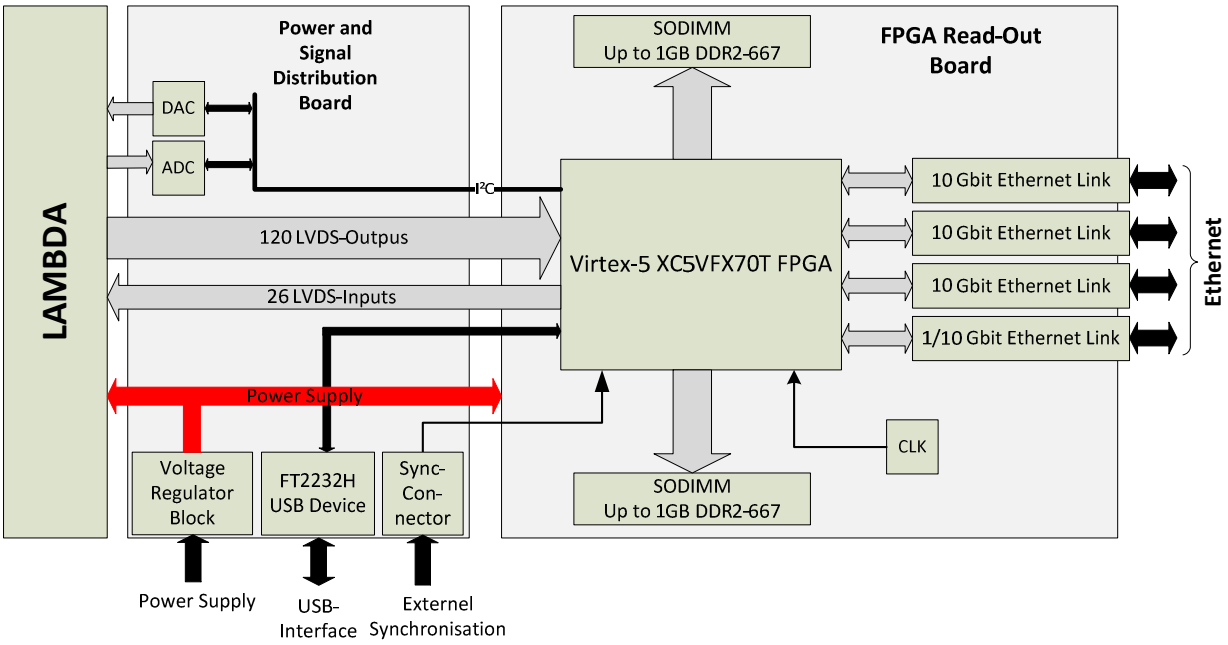


Abbildung 15: Blockschaltbild des Detektors

2.1.8 Large Arrea Medipix Based Detektor Array (LAMBDA)

Das LAMBDA Modul selbst besteht aus 12 Medipix3 Chips, die entsprechend der Abbildung 16, in 2 Reihen von jeweils 6 Chips angeordnet sind. Durch diese Anordnung der Chips ergibt sich eine Auflösung des Detektors von 512x1536 Pixel.

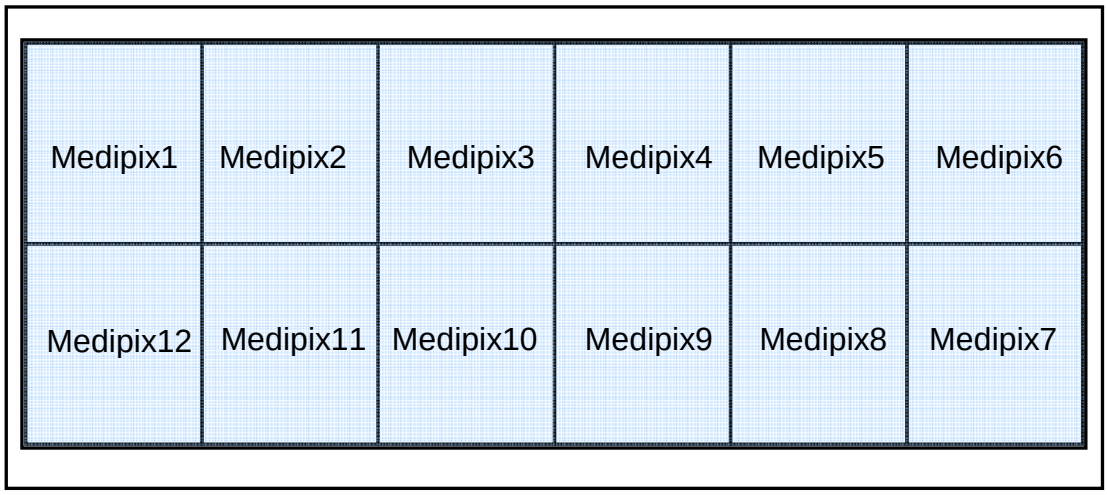


Abbildung 16: LAMBDA-Modul, Ansicht von Oben

Wie bereits beschrieben, verfügt jeder der 12 Medipix3 Chips über acht Eingänge und 10 Ausgänge. Somit ergibt sich eine große Anzahl von I/Os die mit dem FPGA des Readout Bordes verbunden werden müssen. Um die Anzahl der IO's zu minimieren, sind die Eingänge der Chips, bis auf die „Enable-Inputs" als LVDS Multi-Drop Bus entsprechend Abbildung 17 beschaltet. Wobei der LVDS-Treiber mit „D" und die LVDS-Empfänger mit „R" bezeichnet sind.

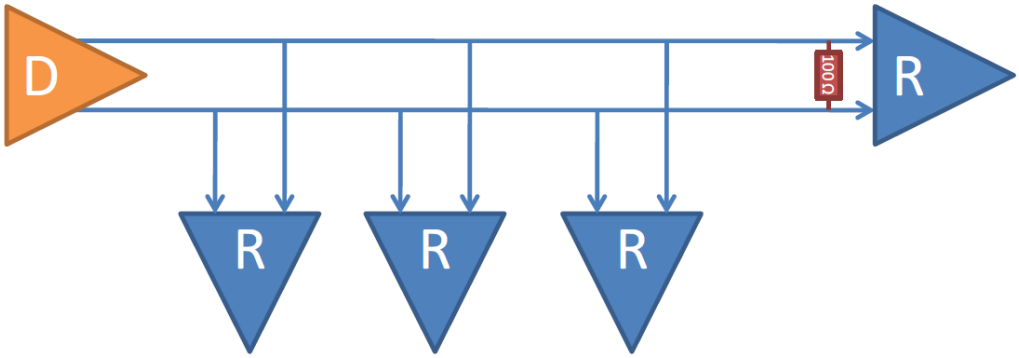


Abbildung 17: LVDS Multi-Drop Bus

Die Abbildung 18 stellt beispielsweise die Beschaltung der LVDS-Eingänge von zwei Medipix3 Chips als LVDS Multi-Drop Bus dar. Das LAMBDA-Modul verfügt über zwei Blöcke mit jeweils sechs Chips, deren Eingänge nach diesem Schema beschaltet sind. Die Adressierung der Chips erfolgt durch die „EnableIn“ Eingänge deren Verbindung mit dem FPGA als direkte Punkt zu Punkt Verbindung realisiert ist. Durch die Anwendung der Multi-Drop Beschaltung wurde die Anzahl der LVDS-Eingänge von 92 (8 Eingänge x 12 Chips) auf 26 differentiale Paare (7 Eingänge x 2 Blocks + 12 EnableIn) reduziert.

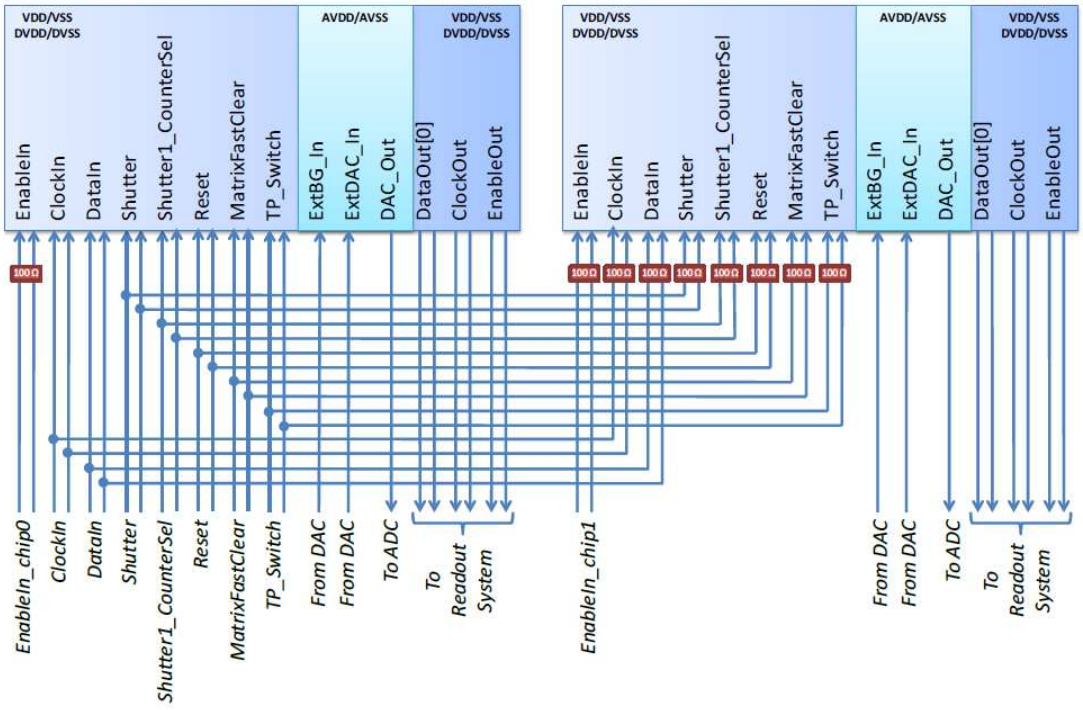


Abbildung 18: Beschaltung von zwei Medipix3 Chips. Quelle [1] S.52

Um die Maximale Readout Geschwindigkeit zu erreichen, müssen alle 12 Chips parallel ausgelesen werden. Deshalb sind die Ausgänge aller 12 Chips als Punkt zu Punkt Verbindung ausgeführt, wodurch die Anzahl der Ausgänge des gesamten LAMBDA-Moduls 10 x 12 = 120 differentiale Paare beträgt.

2.1.9 PFGA Readout Board

Das FPGA Readout Board ist die zentrale Einheit des Detektors die über einen Xilinx XC5VFX70T FPGA der Virtex-5 Familie verfügt. Die wichtigsten Parameter des FPGAs sind in der Tabelle 3 zusammengefasst die dem Datenblatt der Virtex5 FPGA-Familie „DS100“ entnommen wurden.

Device	Configurable Logic Blocks (CLBs)			DSP48E Slices[2]	Block RAM Blocks			CMTs[4]	PowerPC Processor Blocks	Endpoint Blocks for PCI Express	Ethernet MACs[5]	Max RocketIO Transceivers[6]		Total I/O Banks[8]	Max User I/O[7]
	Array (Row x Col)	Virtex-5 Slices[1]	Max Distributed RAM (Kb)		18 Kb[3]	36 Kb	Max (Kb)					GTP	GTX		
XC5VTX150T	200 x 58	23,200	1,500	80	456	228	8,208	6	N/A	1	4	N/A	40	20	680
XC5VTX240T	240 x 78	37,440	2,400	96	648	324	11,664	6	N/A	1	4	N/A	48	20	680
XC5VFX30T	80 x 38	5,120	380	64	136	68	2,448	2	1	1	4	N/A	8	12	360
XC5VFX70T	160 x 38	11,200	820	128	296	148	5,328	6	1	3	4	N/A	16	19	640
XC5VFX100T	160 x 56	16,000	1,240	256	456	228	8,208	6	2	3	4	N/A	16	20	680
XC5VFX130T	200 x 56	20,480	1,580	320	596	298	10,728	6	2	3	6	N/A	20	24	840
XC5VFX200T	240 x 68	30,720	2,280	384	912	456	16,416	6	2	4	8	N/A	24	27	960

Tabelle 3: Eigenschaften der Virtex5 VFX Familie. Quelle [23] S.2

Das FPGA Readout Board verfügt über 148 LVDS-I/Os die zur Verbindung mit dem LAMBDA-Modul vorgesehen sind. Des Weiteren sind auf dem Readout Board zwei 200-Pin Slots zum Einbau von SODIMM Module vorhanden. Aus Kosten- und Verfügbarkeitsgründen werden die Kingston „KVR667D2S5/1G“ SODIMMs verwendet. In der Abbildung 19 sind die beide SODIMM-Slots sowie die entsprechenden SODIM-Module dargestellt.

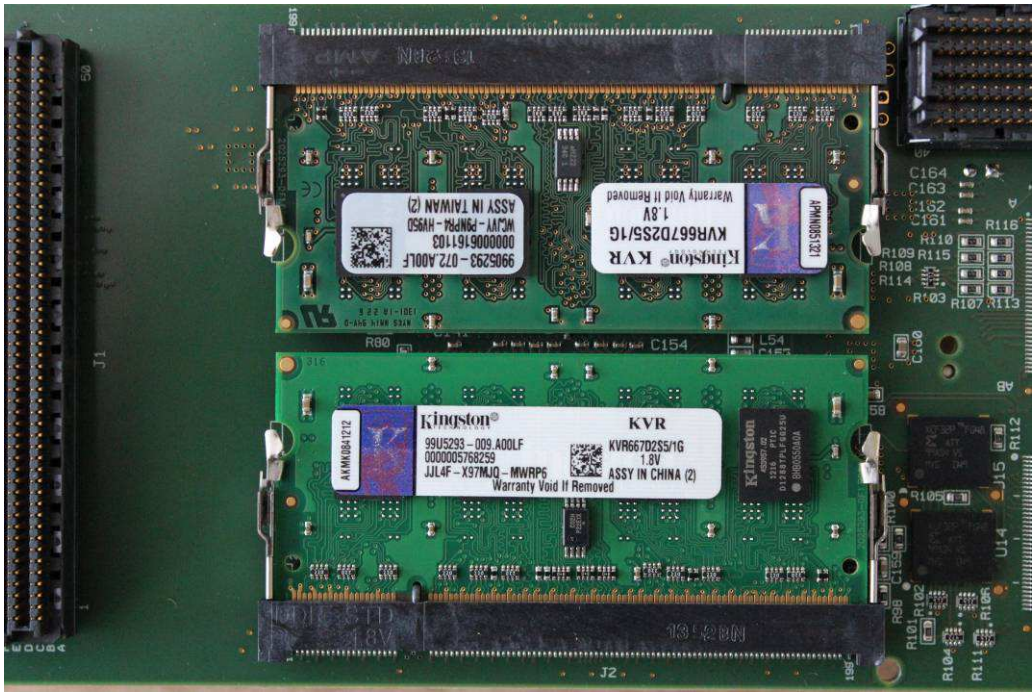


Abbildung 19: SODIMMs des Readout Bordes

Für die Kommunikation des Detektor-Systems mit dem PC über Ethernet sind vier SFP[16]-Slots vorgesehen, die es erlauben, entweder vier 10Gigabit SFP-Transceiver oder drei 10Gigabit- und einen 1Gigabit SFP-Transceiver zu verwenden. Die SFP-Transceiver sind eine neue Generation von modularen Transcivern die ein optisches oder elektronisches Medium nutzen. Die SFP-Transceiver gehören zum Bitübertragungsschicht (Physical Layer) des OSI-Referenzmodels und beinhalten einen Media Dependent- als auch Media Independet Interface. [7] S.22 Die beiden SFP-Transceiver Gruppen der 10G-Transceiver als auch der 1G-Transceiver nutzen ein SGMII[17]-Interface für die Anbindung an die darüber liegende MAC[18]-Instanz.

2.1.10 Power und Signal Distribution Board

Das Power und Signal Distribution Board in der Abbildung 15 dient als Spannungsversorgungs- und Signal-Inteface Board. Dieses Board verfügt über mehrere Spannungsregler zur korrekten Spannungsversorgung des Readout Boardes sowie des LAMBDA-Moduls. Des Weiteren erfolgt über dieses Board die Verbindung von allen LVDS-I/Os des LAMBDA-Moduls mit dem FPGA Readout Board. Zusätzlich verfügt das Power und Signal Distribution Board über Analog/Digital- sowie Digital/Analog Umsetzer für die Ansteuerung der analogen Eingänge der Medipix3 Chips. Für die Serielle Kommunikation verfügt das Board über einen FT2232H USB-Controller der als zweifache virtuelle RS232-Schnittstelle verwendet werden kann. Des Weiteren beinhaltet das Power und Signal Distribution Board einen zusätzlichen Steckverbinder zum Anbinden der externen Signalquellen für die Synchronisation des Detektors bei den Experimenten mit weiteren Subsystemen.

[16] Small Form-Factor Pluggable. Quelle [7, S.22]

[17] Serial Gigabit Media Independent Interface. Quelle [7, S.17]

[18] Media Access Control. Quelle [7, S.11]

3 Präzisierung der Aufgabenstellung

Abgrenzung der Entwicklungsaufgabe

Das Lambda ist ein umfangreiches Projekt, welcher in Form einer Kollaboration zwischen zwei Entwicklungsabteilungen des Deutschen Elektronen Synchrotrons (DESY) und der Technischen Universität München (TUM) entwickelt wird. Dabei werden als eigene Entwicklung das PowerPC 440-basierte eingebettete System, die Software des PowerPC 440 Prozessors sowie Firmware-Komponenten zum Takten des Lambda-Moduls realisiert. Die im vorherigen Kapitel vorgestellte Hardware des Detektors wurde für die Durchführung dieser Arbeit zur Verfügung gestellt. Des Weiteren werden in dieser Diplomarbeit auch Firmware-Komponenten eingesetzt die von den Kollaborationspartnern entwickelt und zur Verfügung gestellt wurden. Dazu zählt die von der TUM entwickelte Parallel-Readout Firmware-Komponente sowie ein 10 Gigabit Netzwerk-Stack das von einer DESY-Partnerabteilung in das Firmware Projekt integriert wurde.

4 Das Detektor Systemkonzept

In diesem Kapitel wird das FPGA Firmware Konzept des Detektors dargestellt, der Vergleich der vorhandenen Mikroprozessoren durchgeführt und eine Entscheidung zur Auswahl des passenden Prozessors getroffen. Dazu wird für das Verständnis des Firmware-Konzeptes die Funktionsweise der wichtigsten Firmware-Komponenten erläutert.

Das Firmware Konzept

Die gesamte Firmware kann in drei wesentliche Einheiten unterteilt werden. Als zentrale Einheit, die der Steuerung des gesamten Detektors dient, wird ein Mikroprozessorbasiertes eingebettetes System verwendet. Als zweite Einheit wird eine Readout Komponente zum Auslesen der Matrixdaten implementiert. Die dritte und letzte Einheit ist einen 10Gbit/s Ethernetlink der für die Hi-Speed Datenübertragung der Detektordaten verwendet wird.

Das eingebettete System das für die Steuerung des Detektors verwendet wird, soll folgende Funktionen erfüllen:

- Ansteuerung der Eingänge der Medipix3 Chips
- Ansteuerung der Firmware Komponenten
- Serielle Datenkommunikation mit den Medipix3 Chips
- TCP/IP[19]-Basierte Datenkommunikation mit PC mittels Gigabit Ethernet
- Realisierung des LocalLink Protokolls zum Übertragen der Matrixdaten der Medipix3 Chips über DMA des eingebetteten Prozessor Block in das DDR2-SDRAM

Der auf dem Readout Board verwendeten Xilinx XC5VFX70T FPGA verfügt über einen eingebetteten PowerPC440 Prozessorblock der als zentrale Einheit des Steuerungssystems verwendet werden kann, des Weiteren bietet das Xilinx EDK[20] einen Microblaze Softprozessor als IP-Core an, welcher auch für die oben genannten Aufgaben geeignet ist. So gesehen bieten sich hier zwei Varianten für die Konzeption

[19] TCP-Transmission Control Protocol. Quelle [7, S.10]

[20] Embedded Development Kit - Eine integrierte Entwicklungsumgebung für die Entwicklung von eingebetteten Verarbeitungssystemen. Quelle [8]

der Steuerungseinheit an: eine PowerPC-basierte Steuerungseinheit und als zweite Variante eine Microblaze-basierte Steuerungseinheit. Um die Auswahl des Mikroprozessors für die Steuerungseinheit zu erleichtern werden im Folgenden die Eigenschaften und Parameter der beiden zur Verfügung stehenden Mikroprozessoren aufgelistet und mit einander verglichen.

4.1.1 PowerPC440 Prozessor

Bei dem in Virtex5 FPGA eingebettetem PowerPC 440 Prozessor handelt es sich um eine erweiterte „IBM Book-E“ Architektur. [3] S.17 Der PowerPC 440 ist ein 32-Bit Reduced Instruktion Set Computer (RISC) CPU[21]. Er ist ein superskalare Prozessor, welches über einen „Dual Issue Instruction Unit“ und drei „Exeqution Piplenes“: Complex Integer Pipeline, Simple Integer Pipeline und Load/Store-Pipeline verfügt. Somit wird durch die Nebenläufigkeit oder dem sogenannten „Instruktion Level Parallelism“ einen CPI[22] Wert von 0,5 erreicht. Das Blockschaltbild des PowerPC 440 Prozessors ist in der Abbildung 20 dargestellt.

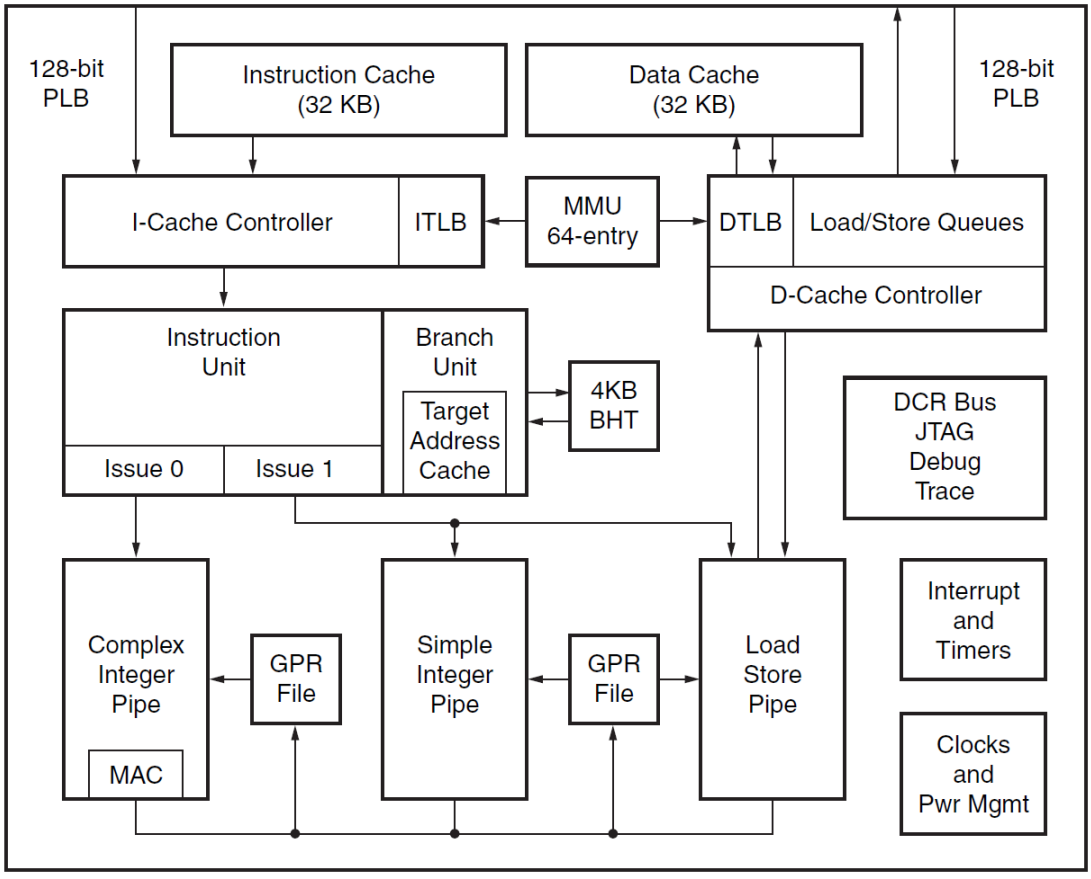


Abbildung 20: Blockschaltbild des eingebetteten PowerPC440 Prozessors. Quelle [3] S.20

[21] Central Processing Unit

[22] Clocks per Instruktion

Jede Pipeline verfügt über vier Bearbeitungsphasen wobei drei vorhergehende Phasen durch die Instruktion-Unit ausgeführt werden. Insgesamt ergibt sich dadurch eine siebenstufige Verarbeitungseinheit. Jede der drei Pipelines hat einen direkten Zugriff auf einen „General Purpose Register Block“ (GPR) welcher 32x32-Bit „Arbeitsregister“ beinhaltet. Um die Performance der Verarbeitungseinheit zu erhöhen und die Zugriffskonkurrenz zwischen den Pipelines zu vermeiden, verfügt der PowerPC440 Prozessor über zwei GPR-Blöcke, wobei ein GPR-Block komplett für die Complex Integer Pipeline vorgesehen ist. Der Zweite wird sowohl von der Simple Integer- als auch Load/Store Pipeline verwendet. Der Inhalt der beiden GPR-Blöcke wird automatisch bei jeder Änderung synchronisiert. [9] S.32

Die Instrukction-Unit des PowerPC440 Prozessors führt Instruction-Fetch, Instrction-Decode und die Ausgabe von zwei Instruktionen per Takt gleichzeitig aus. Daher erreich der PowerPC440 Prozessor einen CPI-Wert von 0.5. Des Weiteren verfügt die Instruction-Unit über eine dynamische Sprungvorhersage (Dynamic Branch Prediction) mit einer Branch History Table (BHT) und internem Speicher für die letzten Sprungziele (Target Address Cache). Durch diesen Mechanismus werden die mittlere Wartezeiten bei der Programmausführung mittels Phasenpipelining verkürzt. [9] S.30

Der PowerPC440 Prozessor verfügt über unabhängige 32Kbyte Cache-Speicher für Daten und Instruktionen die durch Cache-Controller verwaltet werden. Dies ermöglicht dem Prozessor einen sehr schnellen Zugriff auf benötigte Befehle und Operanden soweit diese im Cache vorhanden sind. Beide Cache-Controller sind mit den Processor Local Buses (PLBs) verbunden. Der PowerPC440 Prozessor verfügt über drei unabhängigen 128-Bit PLB-Interfaces. Jedes PLB-Interface beinhaltet einen 36-Bit Adressbus und einen 128-Bit Datenbus [10]. Wobei ein PLB-Interface zum Lesen von Instruktionen und die zwei anderen zum Lesen und Schreiben der Daten vorgesehen sind. Die 36-Bit Adresse wird durch die Memory Management Unit (MMU) in Folge eines Übersetzungsprozesses aus einer effektiven 32-Bit Adresse, die vom Prozessor Core als Instruction-Fetch oder Load/Store Adresse verwendet wird, als physikalische Adresse erzeugt. [9] S.32

Der PowerPC440 Prozessor in der Xilinx Implementierung kann maximal mit bis zu 550MHz Taktfrequenz, je nach Speed Grade des FPGAs, betrieben werden und schneidet im Dhrystone-Benchmark[23] mit 2 DMIPS[24]/MHz ab. Somit erreich er bis zu 1100 Millionen Instruktionen pro Sekunde. [11]

[23] Der Dhrystone-Benchmark führt verschiedene Integer- und String-Operationen durch, jedoch keine Fließkomma-Arithmetik.

[24] Million Instruction Per Secod nach Dhrystone-Benchmark

4.1.2 Microblaze Prozessor

Der Microblaze ist eine 32-Bit RISC Soft-Prozessor welcher für die Implementierung in Xilinx-FPGAs optimiert ist. Dank seiner Optimierung für Xilinx-FPGAs benötigt er je nach Konfiguration nur zwischen 700 und 2000 Slices [12]. Der Microblaze wird als IP-Core von Xilinx angeboten und ist ein Bestandteil des Xilinx XPS[25]. Das Blockschaltbild des Microblaze-Prozessors ist in der Abbildung 21 dargestellt.

Ähnlich wie der PowerPC440 verfügt er über unabhängigen Daten- und Instruktion-Businterfaces. Wobei zum Anbinden eines Speichers der Instruction Local Memory Bus (ILMB) und der Data Local Memory Bus (DLMB) vorgesehen sind. Dadurch zeichnet sich eine typische Harvard-Architektur des Microblaze- Prozessors aus, welche es dem Prozessor erlaubt gleichzeitig auf Daten- und Programm-Informationen zu zugreifen. Die Ankopplung der Prozessor-Peripherie erfolgt wiederum mittels der oben genannten unabhängigen PLB Bussystemen.

Optional kann der Microblaze durch eine Memory Management Unit (MMU) sowie Instruction- und Daten-Caches erweitert werden. Ähnlich wie PowerPC440 beinhaltet der Microblaze einen 32x32Bit General Purpose Register Block (GPR).

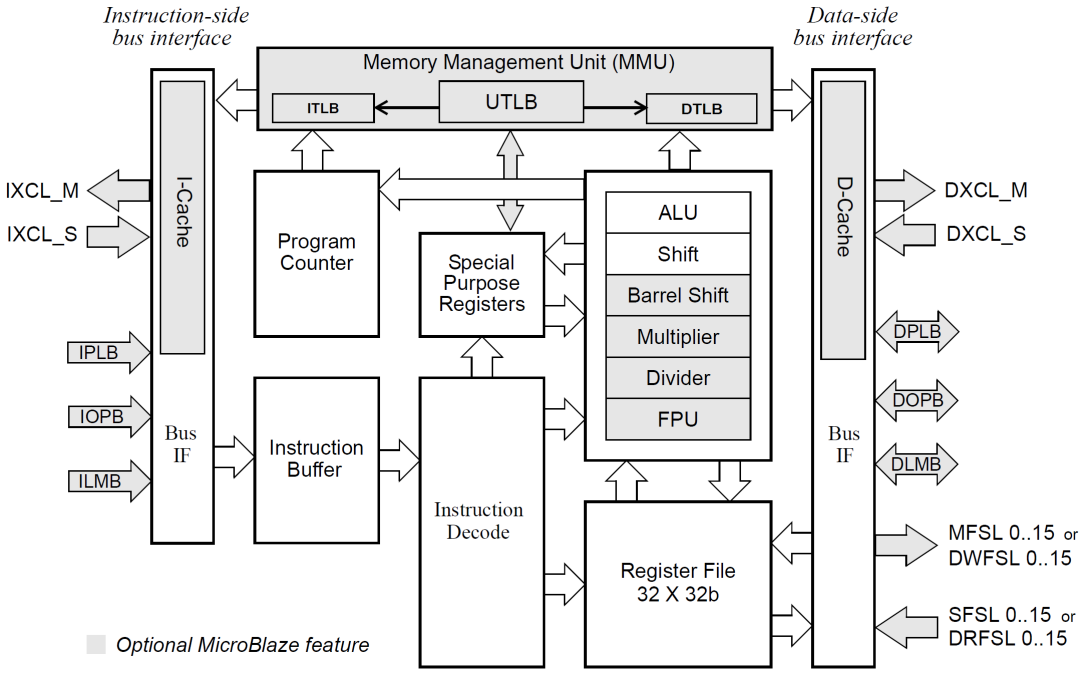


Abbildung 21: Block Schaltbild des Microblaze-Prozessors. Quelle [12] S.9

Er verfügt über eine Instruktion-Execution Pipeline mit drei Bearbeitungsphasen: Instruction-Fetch, Instruction-Decode, Instruction-Execute. Optional kann die Verar-

[25] Xilinx Platform Studio

beitungseinheit des Prozessors um eine weitere Memory-Access Phase und eine Writeback-Phase erweitert werden. Quelle [12] S.50

Die Anzahl der Taktzyklen die für die Ausführung der meisten Instruktionen benötigt werden entspricht etwa der Anzahl der Pipeline-Phasen. Deshalb schafft es der Microblaze-Prozessor bei jedem Taktzyklus die Bearbeitung von jeweils einer Instruktion fertig zu stellen. Somit erreicht er einen CPI-Wert von 1,03 bis 1,38 je nach Anzahl der Pipeline-Phasen und Typen der auszuführenden Instruktionen. Der Microblaze-Prozessor kann deshalb als skalarer Prozessor bezeichnet werden.

Der Microblaze-Prozessor kann mit einer maximalen Taktfrequenz von 150MHz auf den Virtex-5 FPGAs betrieben werden und erreicht im Dhrystone-Benchmark eine Performance, je nach FPGA-Typ und Konfiguration des Prozessors, von 1,03 bis 1,38 DMIPS/MHz [13]. Somit schafft der Microblaze-Prozessor etwa 300 Millionen Instruktionen pro Sekunde.

4.1.3 Vergleich und Entscheidungsfindung

In der Tabelle 4 sind die etnscheidenden Parameter der in Frage kommenden Prozessoren (PowerPC440 und Microblaze) zusammengefasst. Daraus ist es ersichtlich, dass der PowerPC440 Prozessor wesentlich Leistungsfähiger ist als der Soft-Core Microblaze. Mit seinen 1100 DMIPS ist er etwa um Faktor fünf schneller als der Microblaze-Prozessor. Des Weiteren ist er ein Bestandteil des, als dedizierte Hardware integrierten Prozessor Blocks, welcher ihn um vier DMA-Kanäle, einen Memory-Controller-Interface sowie drei weitere Bussysteme: einen Master Processor Local Bus (MPLB) und zwei Slave Processor Local Buses (SPLB) erweitert.

CPU	Max. Taktfrequenz [MHz]	DMIPS bei max. Taktfrequenz	Clocks per Instruction	Verarbeitungsbreite [Bit]
PowerPC440	550	1100	0,5	32
Microblaze	150	154,5-207	1,03-1,38	32

Tabelle 4: Gegenüber Stellung der vorhandenen Mikroprozessoren

Der in Kapitel 4.2.2 vorgestellte Microblaze-Prozessor hat den Vorteil, dass man ihn auf jedem Xilinx FPGA implementieren und an seine Bedürfnisse anpassen kann. Mit dem Microblaze-Prozessor wäre die Detektor-Firmware auf viele Xilinx FPGA portierbar. Er kann nur noch mit einer maximalen Taktfrequenz von 150MHz auf dem Virtex-5 betrieben werden und aufgrund des Aufbaus seiner Verarbeitungseinheit erreicht er maximal einen CPI-Wert von 1,38 und daraus resultierenden 207 DMIPs.

Allerdings ist die geringere Geschwindigkeit ein großer Nachteil, da ein TCP/IP-Basierte Datenkommunikation mit dem PC realisiert werden muss, wofür einen TCP/IP-Stack[26] benötigt wird, deren Protokolle (TCP, IP, etc.) durch den Prozessor realisiert werden müssen. Je höher also die Verarbeitungsgeschwindigkeit des Prozessor-Systems ist, desto besser wird die Performance der Datenübertragung- und allgemein des gesamten Detektor-Systems sein. Da der Detektor explizit für Hochgeschwindigkeitsanwendungen entwickelt wurde ist der PowerPC440 Prozessor hier die richtige Wahl. Des Weiteren würde der Microblaze-Prozessor bis zu 2000 FPGA-Slices benötigen, wohingegen der PowerPC440 Prozessor als dedizierter Hardware keine PFGA-Ressourcen verbraucht. Aufgrund dieser beiden vorher genannten Kriterien fiel die Entscheidung den PowerPC440 Prozessor als Zentrale Einheit der Detektor-Steuerung zu verwenden.

4.1.4 Aufbau und Grundkomponente der Firmware

In der Abbildung 22 ist das Blockschaltbild der Detektor-Firmware dargestellt. Als zentrale Steuereinheit wird ein PowerPC440 basiertes eingebettetes System verwendet.

Die Anbindung der Peripherie-Komponenten an den Prozessor-Block erfolgt mittels eines Master Processor Local Bus (MPLB). Alle verwendeten Peripherie-Komponenten bis auf „Lambda_LL_Interface“ sind IP-Cores die als Bestandteile des XPS[27] zur Verfügung stehen.

Als Arbeitspeicher steht dem PowerPC440 Prozessor einer der beiden SODIMM-Module zur Verfügung die, auf dem FPGA-Board vorhanden sind. Das SODIMM-Modul wird mit Hilfe eines Memory-Controllers an das Memory-Controller-Interface (MCI) des Prozessor-Blocks angeschlossen.

4.1.4.1 Ansteuerung der LAMBDA I/Os und Firmware-Komponenten

Zum Ansteuern der I/O’s der Medipix3 Chips werden drei XPS_GPIO IP-Cores verwendet. Wobei GPIO für General Purpose Input Output [14] steht. Dabei handelt es sich um die bidirektionale Ports die durch den PLB vom Prozessor gesteuert werden und es erlauben Binärinformation auszugeben bzw. einzulesen. Die Portbreite der GPIOs ist einstellbar und kann zwischen 1 und 32Bit gewählt werden.

[26] Durch OSI-Referenzmodel definierter Protokollstapel zu Übertragung der Daten via Ethernet.

[27] Xilinx Platform Studio

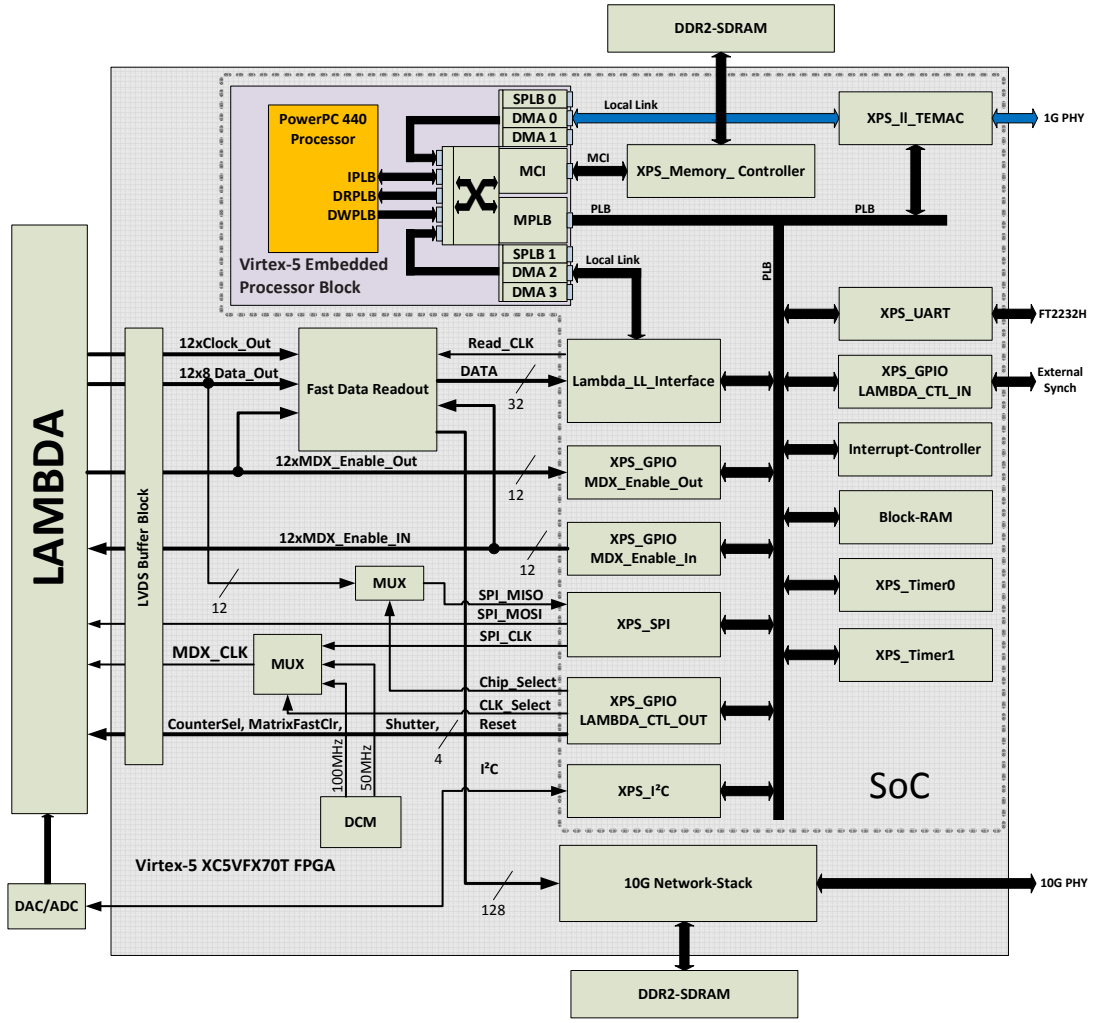


Abbildung 22: Detektor-Firmware-Konzept

Zum Aktivieren und Deaktivieren der Medipix3 Chips durch die „Enable_In“ Eingänge wird der als „MDX_Enable_In“ bezeichneten GPIO-Port verwendet. Deren Bitbreite wird entsprechend der Anzahl der Midipix3 Chips des LAMBDA-Moduls, auf 12 eingestellt.

Die Ankopplung der 12 „Enable_Out“ Ausgänge an das Steuerungs-System erfolgt mittels des „MDX_Enable_Out“ GPIO-Ports. Die „Enable_Out“ Ausgänge der Medipix3 Chips werden zum Quittieren der ausgeführten Chip-Operationen verwendet.

Für die Ansteuerung der Reset, Shutter, CounterSelCRW und MatrixFastClear Chip-Eingänge werden die „LAMBDA_CTL_OUT“ GPIO-Ports verwendet. Des Weiteren übernimmt dieser GPIO-Port die Ansteuerung der Firmware-Internen Multiplexer die zum Umschalten der Taktsignale und Datenausgänge der Medipix3 Chips dienen.

Durch den „LAMBDA_CTL_IN“ GPIO-Port erfolgt die Anbindung der externen Signalquellen für die Synchronisation des Detektors bei den Experimenten mit weiteren Subsystemen.

4.1.4.2 Serielle Daten-Kommunikation mit den Medipix3 Chips

Das Laden des OMR- und DAC Register sowie der Pixel-Matrix der Chips erfolgt bitseriell durch die „DataIn“-Eingänge. Diese serielle Kommunikation mit den Medipix3 Chips erfolgt mit Hilfe eines Serial Peripheral Intarfaces (SPI), welcher durch das XPS-SPI[28] Core repräsentiert wird. Die Übertragung der Daten vom Prozessor zum Medipi3 Chip findet über die SPI_MOSI[29] Leitung statt, welche parallel an die „DataIn“ Eingänge aller 12 Medipix3 Chips angeschlossen ist. Zum Auslesen der Medipix3 Register durch die „Data_Out[0]“ Ausgänge wird auch der XPS_SPI verwendet. Hierfür werden die „Data_Out[0]“ Ausgänge der Chips mit Hilfe eines Multiplexers an die MISO-Leitung des SPI-Cores angeschlossen. Die Ansteuerung des Multiplexers erfolgt wie oben beschrieben durch den „LANBDA_CTL_OUT“ GPIO-Port.

4.1.4.3 Das Lesen der Pixel-Matrix der Medipix3 Chips

Das Lesen der Pixel-Matrix der Medipx3 Chip erfolgt parallel durch alle acht Data-Out[0:8] Ausgänge, wobei alle 12 Medipix3 Chips zeitgleich parallel ausgelesen werden. Somit sind es insgesamt 96 LVDS-Datenausgänge die, an die Fast-Data-Readout Komponente angeschlossen sind. Die Fast-Data-Readout Komponente wurde von der TU-München in Kollaboration für das LAMBDA-Projekt entwickelt. Deshalb wird diese Firmware-Komponente in dieser Arbeit als „Blackbox“ betrachtet. Die Fast-Data-Readout Komponente hat die Aufgabe die 96 seriellen Datenströme zu parallelisieren und durch die Daten-Interfaces in einem geordneten Format auszugeben. Die Fast-Data-Readout Komponente verfügt über zwei parallele Daten-Interfaces: ein 32Bit- und ein 128Bit-breites, die in Abhängigkeit von der eingestellten Auslesegeschwindigkeit und des gewählten Ethernet-Interfaces verwendet werden.

Zum Übertragen der Matrixdaten zum Server sind zwei Varianten vorgesehen:

Variante 1: Übertragung der Daten mittels Gigabit Ethernet

Bei dieser Variante werden die Matrixdaten des LAMBDA-Moduls durch den DMA-Kanal des eingebeteten Prozessorblocks in dem, dem Prozessor zugeordneten DDR2-Memory zwischengespeichert und anschließend mittels Gigabit Ethernet unter Verwendung eines TCP/IP-Protokolls sicher zum PC übertragen.

Entsprechend den Messergebnissen des „Praxisprojektes II“ [7] S.34, in dem das ähnliche eingebettete System zum Übertragung der Daten mittels TCP/IP Verwendet wurde, beträgt die maximal erreichbare Übertragungsbandbreite ca. 800 Mbit/s. Die Größe eines LAMBDA-Bildes beträgt insgesamt 1180032 Byte und setzt sich aus folgendermaßen zusammen:

[28] XPS Serial Peripheral Interface (SPI) (v2.02a). Quelle [15]

[29] Master Output Slave Input

$$256\,Pixel \times 256\,Pixel \times 12\,Bit = 786432\ Bit$$

Gleichung 1: Bildgröße eines Medipix3 Chips

$$786432\ Bit \times 12\,Chips + 3072\,Bit = 9440256\ Bit \Rightarrow \frac{9440256\ Bit}{8} = 1180032\ Byte$$

Gleichung 2: Bildgröße eines LAMBDA-Moduls

wobei die zusätzlichen 3072 Bit in der Gleichung 3 als Header-Information durch die Fast-Data-Readout Komponente jedem Bild hinzugefügt wird.

Mit Hilfe der oben genannten Werte der Übertragungsbandbreite und Bildgröße, lässt sich die theoretische Bildwiderholungsfrequenz annähernd berechnen. Allerdings muss hier beachtet werden, dass sich der PowerPC440 Prozessor, im Gegensatz zum Praxisprojekt II, nicht nur mit der Übertragung der Daten, sondern auch mit der Ansteuerung der Medipix3 Chip beschäftigt. Des Weiteren nimmt das Auslesen der Pixelmatritzen der Medipix3 Chips auch eine bestimmte Zeit in Anspruch. Daher muß, um die theoretische Bildwiderholungsfrequenz annähernd berechnen zu können, sowohl die Zeit zum Übertragen der Daten über Ethernet als auch zum Auslesen der Pixel-Matrix ermittelt werden.

Die Übertragungszeit eines kopletten LAMBDA-Bildes über das Gigabit Ethernet Interface mit der Bandbreite von 800Mbit/s ergibt sich zu:

$$\frac{9440256\ \ Bit}{800 \times 10^{6}\,Bit\ /\ s} = 0{,}0118\ s \Rightarrow 11{,}8\,ms$$

Gleichung 3: Übertragungszeit eines LAMBDA-Bildes über Ethernet

Zum Auslesen der Pixel-Matrix des Lambda-Moduls werden die Medipix3 Chips und Fast-Data-Readout Komponente mit 50 MHz getaktet und die Matrixdaten der Medipix3 Chips werden jeweils über acht Datenausgänge parallel ausgelesen. Damit ergibt sich die Anzahl der Taktzyklen die zum Auslesen eines LAMBDA-Bildes benötigt werden zu:

$$\frac{9440256\ \ Bit}{8\,Ausgänge\ \ \times 12\,Chips} = 98336\ Taktzyklen$$

Gleichung 4: Anzahl der benötigten Taktzyklen zum Auslesen eines Lambda-Images

Die Auslesezeit eines LAMBDA-Bildes beträgt somit:

$$\frac{98336\,Taktzyklen}{50 \times 10^{6}\,Hz} = 1{,}966\,ms$$

Gleichung 5: Auslesezeit eines LAMBDA-Bildes

Insgesamt ergibt sich damit zum Auslesen und Übertragen eines LAMBDA-Bildes eine Zeit von:

$$1{,}966ms + 11{,}8ms = 13.766ms$$

Gleichung 6: Auslese- und Übertragungszeit eines LAMBDA-Bildes insgesamt

Die maximale Bildwiederholungsfrequenz beim Verwenden eines Gigabit Ethernet-Interfaces beträgt damit:

$$\frac{1}{13{,}766 \times 10^{-3} s} = 72.6 Bilder / s$$

Gleichung 7: Bildwiederholungsfrequenz

Allerdings werden hier die Zeiten, die der PowerPC440 Prozessor zum Ansteuern des LAMBDA-Moduls sowie die Zugriffszeiten auf Peripheriekomponenten und den Speicher vernachlässigt. Deshalb wird die tatsächliche Bildwiederholungsfrequenz etwas niedriger sein und kann nach der Umsetzung des Konzeptes in der Hardware, genau gemessen werden.

Die 50 MHz Taktfrequenz wurde ausgehend von der Übertragungsbandbreite des DMA-Kanals gewählt. Der Grund dafür wird im Kapitel „Taktung des LAMBDA-Moduls und der Readout Komponenten“ näher erklärt.

Zusammenfassend kann gesagt werden, dass diese Readout Variante sich für Experimente mit niedrigen Frame-Raten eignet, bei denen es darauf ankommt, dass die Daten verlustfrei zum PC übertragen werden.

Die Anbindung der Fast-Data-Readout Komponente an den DMA-Kanal des eingebetteten Prozessor Blocks erfolgt mittels des „Lambda_LL_Interfaces“. Mit Hilfe dieser Firmware-Komponente wird der einlaufende Datenstrom unter Verwendung eines Protokolls zur Übertragung durch den DMA-Kanal vorbereitet. Auf den Entwurf dieser Komponente wird im Kapitel „Technische Umsetzung“ näher eingegangen.

Die Übertragung der Matrixdaten zum Server erfolgt wie bereits beschrieben über Gigabit-Ethernet mittels TCP/IP-Protokoll, wobei die zweiten Schicht des Netzwerk-Stapels [7] S.10, der „Data Link“, durch einen, als statische Hardware im FPGA integrierten „Tri-Mode Ethernet Media Access Controller“ oder „TEMAC“ realisiert wurde. Der „TEMAC“ wird durch einen „XPS_LL_TEMAC“ IP-Core [17] in der Firmware repräsentiert. Die oberen Schichten des Netzwerk-Stapels sowie deren IP- und TCP-Protokolle werden durch LwIP-Bibliotheken [7] S.17 als Software des PowerPC440 Prozessors realisiert. Die Anbindung des TEMACS an den DMA-Kanal des eingebetteten Prozessor Blocks erfolgt mittels Local Link Interfaces. Die Konfiguration der Internen Register des IP-Cores erfolgt über den „Processor Local Bus“.

Variante 2: Übertragung der Daten mittels 10 Gigabit Ethernet

Bei der zweiten Variante erfolgt die Übertragung der Matrixdaten nebenläufig zum eingebetteten System mittels 10 Gbit Ethernet-Interfaces unter Verwendung eines UDP-Protokolls. Wobei das 128Bit Daten-Interface der Fast-Data-Readout Komponente verwendet wird.

Die Umsetzung der UDP, IP und MAC Netwerk-Protokolle erfolgt in diesem Fall als in FPGA-Softlogik implementierter 10G Netzwerk-Stapel. Diese Firmware-Komponente wurde ebenfalls durch die Partner-Abteilung bei DESY entwickelt und die genaue Funktionsweise wird hier ebenfalls als „Blackbox“ betrachtet.

Durch den Einsatz eines 10 Gigabit Ethernet Interfaces steht etwa die 10-fache Datenübertragungsbandbreite im Vergleich zum Gigabit-Interface zur Verfügung. Deshalb werden zum Auslesen der Pixel-Matrix die Medipix3 Chips sowie die Fast-Data-Readout Komponente mit 100 MHz Taktfrequenz betrieben. Somit beträgt die Auslesezeit eines LAMBDA-Bildes mit den dafür benötigten 98336 Taktzyklen:

$$\frac{98336\,Taktzyklen}{100\times10^{6}\,Hz}=0{,}983\,ms$$

Gleichung 8: Auslesezeit eines LAMBDA-Bildes mit 100 MHz Taktfrequenz

Damit können über 1000 Bilder/s ausgelesen werden. Für die Übertragung von 1000 Bildern/s über 10G Ethernet wird eine Übertragungsbandbreite von:

$$9440256\,Bit\times1000\,Bilder/s=9440256\times10^{3}\,Bis/s\Rightarrow 9{,}45\,GBit/s$$

Gleichung 9: Benötigte Übertragungsbandbreite für 1000 Bilder/s

benötigt und kann damit durch einen 10 Gigabit Link realisiert werden.

4.1.4.4 Weiteren Firmware Komponenten

Das Signal Distribution Board verfügt über einen FT2232H USB-Controller welcher als zweifacher Virtual-Comport für die Serielle Datenkommunikation mit dem PC verwendet werden kann. Für deren Anbindung an das eingebettete System kommt ein „XPS_UART“ IP-Core zum Einsatz. Dies ermöglicht eine einfache Textausgabe über den Com-Port, der damit zum Debugging des Systems verwendet werden kann.

Das Block-RAM wird zum Booten des PowerPC440 Prozessors benötigt. Diese beinhaltet einen Bootloader welchen der PowerPC440 Prozessor an die Startadresse des Prozessorprogramms während des Boot-Vorganges verlinkt. Das Programm des PowerPC440 Prozessors befindet sich im DDR2-SDRAM und wird durch einen System ACE Controller [18] beim Start des Systems dorthin geladen.

Der Interrupt-Controller dient der Erweiterung des Interrupt-Systems des PowerPC440 Prozessors. Der PowerPC440 Prozessor verfügt selbst nur über einen Eingang für externe Interrupt-Quellen. Um mehrere Interrupt-Quellen an den Prozessor anschließen zu können, wird deshalb der Interrupt-Controller eingesetzt. Als Interrupt-Quellen in diesem Firmware-Design dienen die „MDX_Enable_Out XPS_GPIO“, „XPS_ll_TEMAC“ sowie „XPS_Timer“. Auf das Interrupt-Konzept wird im Kapitel „Technische Umsetzung“ näher eingegangen.

Die beiden Timer „XPS_Timer0“ und „XPS_Timer1“ werden zum Realisieren von Zeitdefinierten Readout Modi verwendet. Des Weiteren wird der „XPS_Timer1“ zum Ermitteln der Übertragungsbandbreite des DMA-Kanals sowie der Ethernet-Schnittstellen eingesetzt.

Durch den „LVDS Buffer-Block“ erfolgt die Anbindung der LVDS I/Os der Medipix3 Chips an die Firmware Komponente. Hierfür beinhaltet der Block 120 LVDS Receiver und 26 LVDS Transmitter.

5 Technische Umsetzung

Dieses Kapitel beschäftigt sich mit der technischer Umsetzung des im Kapitel 4 vorgestellten Firmware-Konzeptes. Es wird die Vorgehensweise bei der Erstellung des PowerPC 440-basierten eingebetteten Systems erläutert sowie auf die wichtigsten Periphere-Komponenten eingegangen. Weiter folgt eine Beschreibung des Taktungskonzeptes des Lambda-Moduls und der Readout-Komponente. Schließlich wird auf das Software-Konzept des PowerPC 440 Prozessors und dessen Umsetzung eingegangen.

Firmware, Entwurf und Implementierung

Für die Entwurf und Implementierung der Detektor-Firmware und deren Komponenten werden folgende Xilinx Werkzeuge eingesetzt:

- ISE 13.4: In der Entwicklungsumgebung Xilinx-ISE[30] wird das komplette FPGA Firmware-Design umgesetzt. Das ISE-Projekt beinhaltet alle Firmware-Komponente, deren Verbindungen und hierarchischen Zusammenhänge. Selbst das eingebettete System wird in dieser Umgebung als Firmware-Komponente instanziiert. Die Top-Entity dieses ISE-Projektes beschreibt alle Ports der Firmware die den FPGA-Pins zugeordnet werden. Des Weiteren erfolgt mit den Tools dieser Umgebung die gesamte FPGA-Konfiguration von der Synthese des VHDL-Codes bis hin zur Erzeugung des Bitsreams.

- XPS 13.4: Das Xilinx Plattform Studio wird zum Erstellen und Konfigurieren des eingebeteten Systems verwendet. Diese wird dann nach der Fertigstellung als Komponente in der ISE-Umgebung instanziiert.

- SDK 13.4: Mit dem Software Development Kit erfolg die gesamte Software Entwicklung des eingebetteten PowerPC440 Prozessors von der Kodierung in System C bis hin zur Erzeugung einer Elif[31]-Datei für die Konfiguration des Prozessors.

In Abbildung 23 die der Quelle [8] entnommen wurde, sind die gesamten Entwicklungs- und Implementierungsabläufe sowie die Zusammenhänge der verwendeten Werkzeuge dargestellt.

[30] Integrated Software Environmen

[31] Executable and Linkable Format

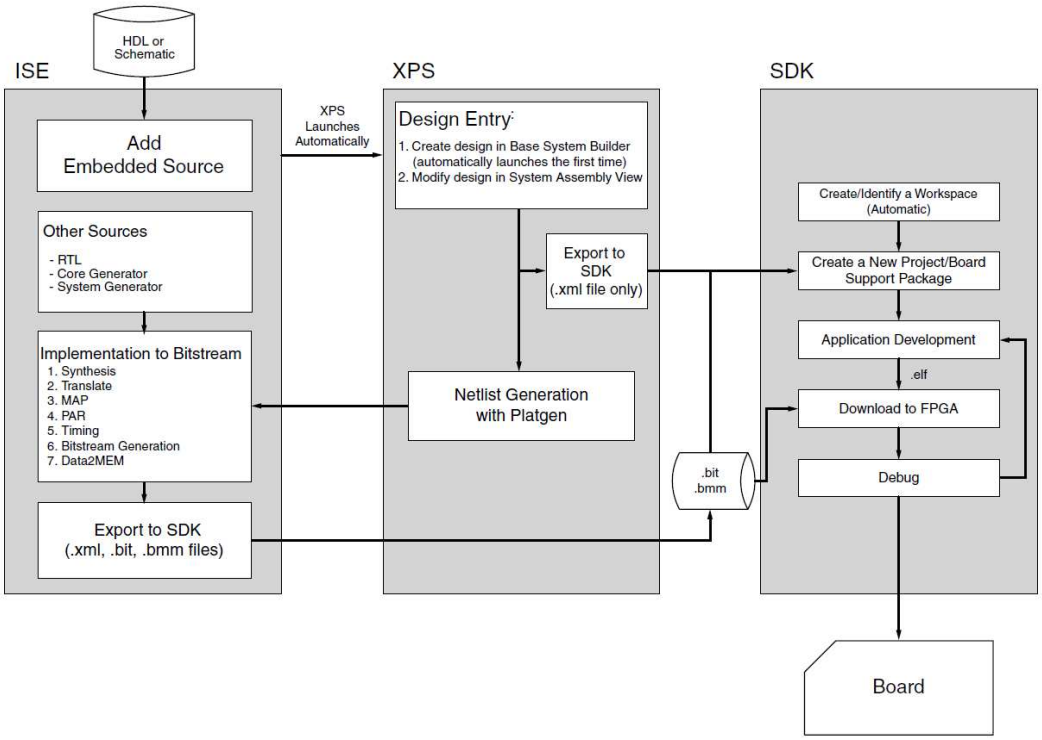


Abbildung 23: Implementation Design Flow. Quelle [8] S.10

Die Abbildung 24 stellt die Verzeichnisstruktur des Projektes dar. Das „LAMBDA_PPC_XPS“ Verzeichnis beinhaltet das PowerPC 440-Basiertes XPS Projekt sowie alle Quelldateien dieses Projektes.

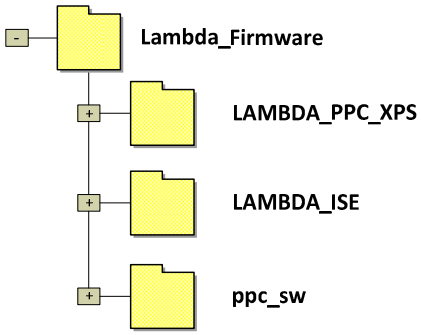


Abbildung 24: Verzeichnisstruktur des Projektes

In dem „LAMBDA_ISE“ Verzeichnis befindet sich das vorher beschriebenes ISE-Projekt. Das gesamten SDK-Software-Projekt des PowerPC 440 Prozessors kann im „ppc_sw“ Verzeichnis gefunden werden. Die in Abbildung 24 dargestellten Verzeichnisse befinden sich auf der beilegenden DVD.

5.1.1 Erstellen und Konfigurieren eines eingebetteten Systems

Das eingebettete System der Detektor-Firmware wurde mit Hilfe eines Base System Builders (BSB) des XPS erstellt. Dieser führt den Design-Entwickler schrittweise von der Auswahl des zu verwendenden Prozessors, dessen Taktung und Debuggingmethode bis hin zur Einbindung der nötigen IP-Cores.

Als Zielplattform würde das bei dem Praxisprojekt II verwendeten Evaluation-Board ML507 ausgewählt. Da das Readout Board über denselben FPGA-Typ der XC5VFX70T und auch die SODIMMs verfügt, werden die passende IP-Cores zur Auswahl gestellt, die dem Design hinzugefügt und an das Bussystem des Prozessors angeschlossen werden können. Abbildung 25 zeigt den „Base System Builder“ und die durch dieses Tool hinzugefügten Design Komponenten.

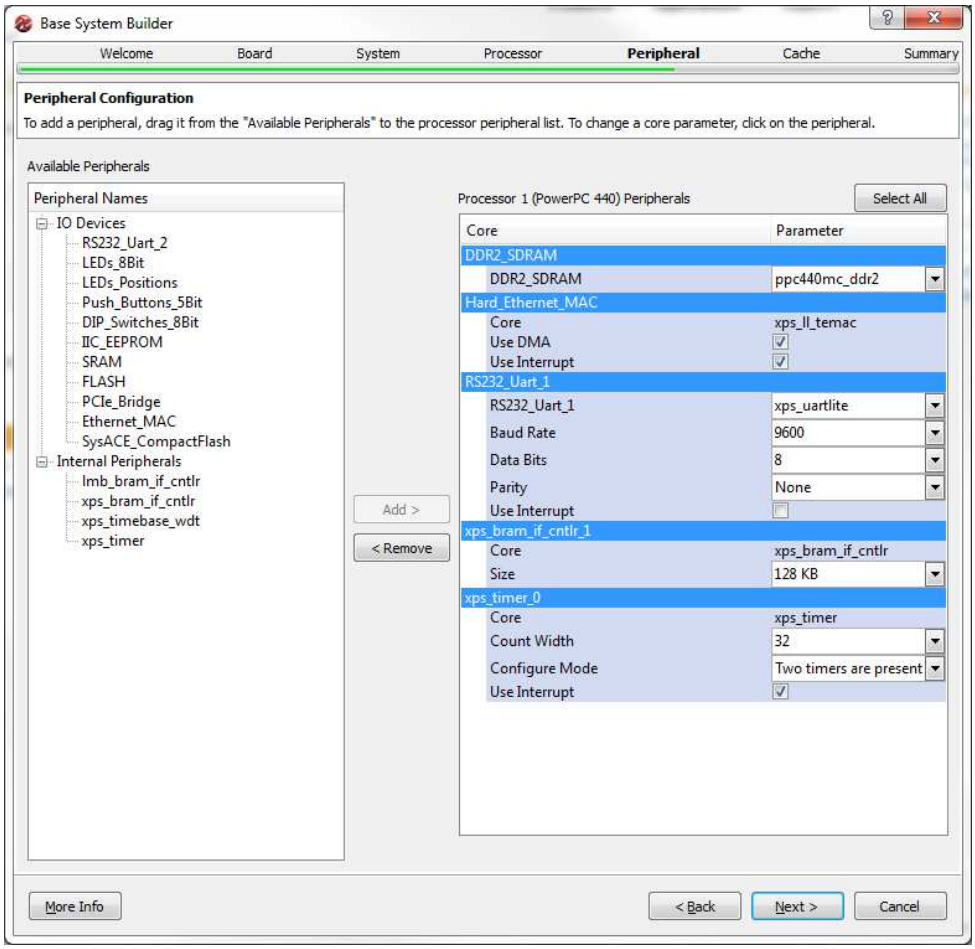


Abbildung 25: Base System Builder

Nach der Fertigstellung des „Base System Builders“ wird das Design des eingebetteten Systems erstellt. Das Design beinhalten einen PowerPC440 Prozessorblock, einen DDR2 Memory Controller (ppc440mc_ddr2) welcher an das „Memory Controller Interface“ (MCI) des Prozessorblocks angeschlossen ist und ist als DDR2_SDRAM bezeichnet wird. Des Weiteren verfügt das System über den „Processor Local Bus“ (PLB) und die folgenden Komponenten die in Form von XPS IP-Cores an dieses PLB angeschlossen sind:

- xps_ll_temac: XPS Local Link Tri Mode Ethernet Media Access Controller IP-Core
- xps_uartlite: XPS RS232 Universal Asynchronous Receiver Transmitter (UART) Lite IP-Core
- xps_bram_if_cntlr: XPS Block-RAM Interface Controller IP-Core
- xps_timer: XPS Timer/Counter IP-Core
- xps_intc: XPS Interrupt Controller.

- xps_proc_sys_reset: XPS Processor Reset Module IP-Core
- xps_clock_generator: XPS Clock Generator IP-Core

Die XPS Interrupt Controller, das XPS Processor Reset Modul und der XPS Clock Generator sind in Abbildung 25 nicht aufgelistet, werden aber automatisch durch den BSB dem Design hinzugefügt.

Die weiteren Design Komponenten: MDX_Enable_Out-, MDX_Enable_In-, LAMBDA_CTL_Out-, LAMBDA_CTL_In XPS_GPIOs und XPS_SPI IP-Cores wurden dem Design nachträglich manuell mit Hilfe der XPS Benutzeroberfläche hinzugefügt und an den PLB des Prozessors angeschlossen. Das Lambda_LL_Interface wird nicht von XPS zur Verfügung gestellt. Dabei handelt es sich um eine eigene Entwicklung auf die in den nachfolgernden Kapiteln eingegangen wird.

Wie bereits beschrieben wurde, sind alle Komponenten des eingebetteten Designs durch den PLB mit dem PowerPC440 Prozessor verbunden. Für die Adressierung der Komponenten über PLB wird für jede Komponente ein Adressraum entsprechend der Abbildung 26 festgelegt:

Bus Interfaces | Ports | Addresses

Instance	Base Name	Base Address	High Address	Size	Bus Interface(s)	Bus Name
ppc440_0's Address Map						
DDR2_SDRAM	C_MEM_BASEADDR	0x00000000	0x3FFFFFFF	1G	PPC440MC	ppc440_0_PPC440MC
MDX_ENB_OUT	C_BASEADDR	0x81400000	0x8140FFFF	64K	SPLB	plb_v46_0
MDX_ENB_IN	C_BASEADDR	0x81420000	0x8142FFFF	64K	SPLB	plb_v46_0
LAMBDA_CTRL_OUT	C_BASEADDR	0x81440000	0x8144FFFF	64K	SPLB	plb_v46_0
LAMBDA_CTRL_IN	C_BASEADDR	0x81460000	0x8146FFFF	64K	SPLB	plb_v46_0
xps_intc_0	C_BASEADDR	0x81800000	0x8180FFFF	64K	SPLB	plb_v46_0
xps_spi_0	C_BASEADDR	0x83400000	0x8340FFFF	64K	SPLB	plb_v46_0
xps_timer_1	C_BASEADDR	0x83C00000	0x83C0FFFF	64K	SPLB	plb_v46_0
xps_timer_0	C_BASEADDR	0x83C20000	0x83C2FFFF	64K	SPLB	plb_v46_0
RS232_Uart_1	C_BASEADDR	0x84000000	0x8400FFFF	64K	SPLB	plb_v46_0
Hard_Ethernet_MAC	C_BASEADDR	0x87000000	0x8707FFFF	512K	SPLB	plb_v46_0
lambda_ll_dma_0	C_BASEADDR	0xC2E00000	0xC2E0FFFF	64K	SPLB	plb_v46_0
xps_bram_if_cntlr_1	C_BASEADDR	0xFFFF8000	0xFFFFFFFF	32K	SPLB	plb_v46_0

Abbildung 26: Address-Map des eingebetteten Systems

Wobei die „Base Address“ die Startadresse eines Speicherbereiches ist welcher den Komponenten zugeordnet wurde. Die „High Address“ definiert dabei die höchste Adresse des Speicherbereiches. Diese Adressräume sind Byte-adressierbar und werden mit Ausnahme des DDR2_SDRAMs und XPS Block-Rams in kleinere 32 Bit Bereiche unterteilt die als Register der Peripherie-Komponenten verwendet werden. Durch diese Register wird die Funktionalität der Komponenten programmiert.

Damit ist das Design des eingebetteten Systems erstellt. Der weitere Entwicklungsprozess besteht in der Konfiguration der Design-Parameter und der Anpassung der I/O-Schnittstellen der Komponenten an die vorhandene Hardware des Detektor-Systems.

Nachfolgend wird ein Überblick über die Definition der ausgewählten Komponenten des Designs und deren Aufbau gegeben.

5.1.1.1 PowerPC440 DDR2 Memory Controller

Der DDR2 Memory Controller für den PowerPC 440 Prozessor hat die Aufgabe, die DDR2 SODIM-Module an das Memory Controller Interface (MCI) des eingebetteten Po-

werPC440 Prozessorblocks anzubinden. Das DDR2 SDRAM spielt eine wichtige Rolle für den Betrieb des Prozessors, in dem sich das Programm des Prozessors befindet. Dieses wird bei der Initialisierung des Systems dorthin geladen. Des Weiteren agiert der DDR2 SDRAM als Arbeitsspeicher des Prozessors welcher auch von den DMA-Engines des Prozessorblocks adressiert wird.

Beim Erstellen des Designs durch den Base System Builder werden die Parameter des DDR2 Memory Controllers für einen MICRON „MT46V16M16_5B“ SODIMM optimiert, da dieses Speichermodul Bestandteil des ML507 Evaluation-Boards ist. Damit das verwendete Kingstone „KVR667D2S5/1G“ SODIM-Modul ordnungsgemäß funktioniert kann mussten die Daten- und Adressbusbreiten des Memory Controllers angepasst werden. In Abbildung 27 sind die passenden Parameter für den oben genannten SODIMM in der MHS[32]-Datei des XPS Projektes aufgelistet. Die weitergehende Beschreibung des MHS-Formates kann dem Kapitel 2 der Quelle [20] entnommen werden.

```
1  BEGIN ppc440mc_ddr2
2   PARAMETER INSTANCE = DDR2_SDRAM
3   PARAMETER C_DDR_BAWIDTH = 3         # Bank Adressbreite
    ..
4   PARAMETER C_DDR_DWIDTH = 64         # Dateninterfacebreite
5   PARAMETER C_DDR_CAWIDTH = 10        # Column-Adressbreite
6   PARAMETER C_NUM_RANKS_MEM = 1       # Anzahl der Ranks
    ..
7   PARAMETER C_DDR_RAWIDTH = 14        # Row-Adressbreite
8   PARAMETER C_DDR_BURST_LENGTH = 4    # Burst-Länge
    ..
9   BUS_INTERFACE PPC440MC = ppc440_0_PPC440MC
    ..
10 END
```

Abbildung 27: Interface Parameter des SODIMMs, definiert durch MHS-Datei

Des Weiteren wurden die Timing-Parameter entsprechend dem Datenblatt [22] des Moduls wie folgt konfiguriert:

```
1  BEGIN ppc440mc_ddr2
2   ..
3   PARAMETER C_DDR_CAS_LAT = 5   # CAS Latency (Clock cycles)
4   PARAMETER C_DDR_TRAS = 40000  # Row Active Time (ps)
5   PARAMETER C_DDR_TRCD = 15000  # Row address to column adress delay (ps)
6   PARAMETER C_DDR_TRFC = 127500 # Refresh to Comand or next Refresh (ps)
7   PARAMETER C_DDR_TRP = 15000   # Row Precharge Time (ps)
8   PARAMETER C_DDR_TRTP = 7500   # Read to Precharge Time (ps)
9   PARAMETER C_DDR_TWR = 15000   # Write Recovery Time (ps)
    ..
10 END
```

Abbildung 28: Timing Parameter des SODIMMs, definiert durch MHS-Datei, Zeitangeben sind in Pikosekunden.

Für die Adressierung des SODIMMs wurde der Adressraum von 1 GByte entsprechend der Abbildung 29 festgelegt.

```
1  BEGIN ppc440mc_ddr2
    ..
2   PARAMETER C_MEM_BASEADDR = 0x00000000 # Startadresse des Speichers
3   PARAMETER C_MEM_HIGHADDR = 0x3FFFFFFF # Endadresse des Speichers
```

[32] Microprocessor Hardware Specification

```
    . .
4   END
```

Abbildung 29: Definition des Adressraums des DDR2-SODIMMs

Die Instanz des DDR2 Memory Controllers beinhaltet auch weitere Parameter deren voreingestellte Werte nicht verändert wurden diese können der vollständigen Auflistung der MHS-Datei auf der beiliegenden DVD unter folgendem Pfad entnommen werden: *„Lambda_Firmware\Lambda_PPC_XPS\system.mhs“*

5.1.1.2 General Purpose Input Output

Mit GPIO stellt Xilinx XPS einen VHDL-Soft-Core zur Verfügung welcher es erlaubt die Daten durch parallele Register auf einfache Weise mit anderen Komponenten oder Systemen auszutauschen. Die Register des Cores sind dabei durch den PLB vom Prozessor adressierbar. In Abbildung 30 ist das Blockschaltbild des GPIOs dargestellt. Die Ausgangsdaten werden durch das „GPIO_DATA“ Register bereitgestellt. Dem „GPIO_DATA“-Register ist ein Tri-State Treiber nachgeschaltet, welches zusammen mit dem „GPIO_TRI“-Register die Datentransferrichtung festlegt. Zum Lesen der binären Information sind die Register „GPIO_DATA_IN“ sowie „READ_REG_IN“ vorgesehen.

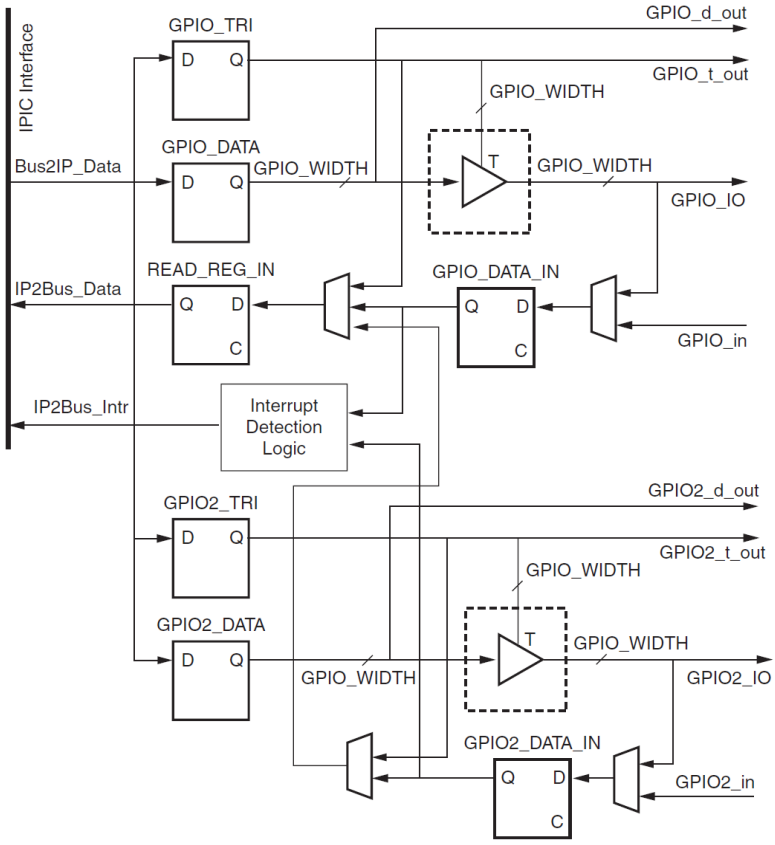


Abbildung 30: Blockschaltbild eines XPS GPIO. Quelle [14] S.3

Der in Abbildung 30 dargestellte GPIO beinhaltet zwei identische Kanäle und eine „Interrupt Detection Logic“. Der zweite Kanal und die „Interrupt Detection Logic“ sind optional

und können durch die Generic-Parameter der Instanz dem Design hinzugefügt oder entfernt werden. Des Weiteren werden die Tri-State Buffer dem IP-Core bei der RTL[33]-Synthese hinzugefügt wenn die bidirektionale GPIO-Ports „GPIO_IO“ verwendet werden. Allerdings ist dies nur dann möglich wenn diese direkt die FPGA Pins treiben, da interne Tri-State Buffer in den FPGAs der Virtex 5 Familie nicht vorgesehen sind.

Die vier Register des GPIOs werden durch die Base-Adresse (C_BASEADDR) und einen Address-Offset entsprechend Tabelle 5 adressiert.

Register Name	Description	PLB Address	Access
GPIO_DATA	Channel 1 XPS GPIO Data Register	C_BASEADDR + 0x00	Read/Write
GPIO_TRI	Channel 1 XPS GPIO 3-state Register	C_BASEADDR + 0x04	Read/Write
GPIO2_DATA	Channel 2 XPS GPIO Data register	C_BASEADDR + 0x08	Read/Write
GPIO2_TRI	Channel 2 XPS GPIO 3-state Register	C_BASEADDR + 0x0C	Read/Write

Tabelle 5: XPS GPIO Register. Quelle [14] S.9

In dem Design des eingebetteten Systems sind vier GPIO Komponenten eingesetzt, die für die Ansteuerung der Lambda I/Os sowie internen Komponenten der Detektor Firmware verwendet werden.

Die „MDX_ENB_IN“ und „LAMBDA_CTRL_OUT“ GPIOs sind als reine Ausgangsports konfiguriert. Die Instanzen dieser GPIOs zeigen die Abbildungen 33 und 34. Diese beiden GPIOs treiben sowohl die LVDS-Transmitter als auch interne Firmware-Signale weshalb die internen Tri-State Buffer nicht verwendet wurden. Mit dem Tri-State Buffer entfällt auch die Verbindung der „GPIO_DATA“-Register mit den „GPIO_DATA_IN“-Register welche durch diese Buffer, entsprechend der Abbildung 30 realisiert wird. Die Verbindung der oben genannten Register ist sehr nützlich, da sich mit ihr Bitmanipulationen des „GPIO-DATA“-Registers realisieren lassen, welche aber sowohl schreibende als auch lesende Zugriffe benötigen. Um dieses Problem zu beseitigen wurden die „GPIO_DATA“- und „GPIO_DATA_IN“-Register manuell durch die Ports der Instanzen mit einander Verbunden. Die Zeilen 8 und 9 der Abbildungen 31 und 32 stellen diese Verbindung dar. Des Weiteren entfällt auch die Notwendigkeit bei der zukünftigen Software Entwicklung die „GPIO_TRI“-Register zu Konfigurieren um die Datentransferrichtung festzulegen, da die gesamte Tri-State Steuerung nicht verwendet wird.

```
1  PORT MDX_ENB_IN_GPIO_O_pin = MDX_ENB_IN_GPIO_IO_O, DIR = O, VEC = [0:11]
   ..
2  BEGIN xps_gpio
3   PARAMETER INSTANCE = MDX_ENB_IN          #Instanzbezeichnung
4   PARAMETER HW_VER = 2.00.a                #Versionsnummer des IP-Cores
5   PARAMETER C_GPIO_WIDTH = 12              # Breite des GPIOs (Bit)
6   PARAMETER C_BASEADDR = 0x81420000        # Baseadresse
7   PARAMETER C_HIGHADDR = 0x8142ffff        # Endadresse
8   BUS_INTERFACE SPLB = plb_v46_0           # Bus Interface
```

[33] Register-Transfer-Level-Synthese

```
9   PORT GPIO_IO_O = MDX_ENB_IN_GPIO_IO_O     # Output Port
10  PORT GPIO_IO_I = MDX_ENB_IN_GPIO_IO_O     # Input Port
11 END
```

Abbildung 31: Definition der Instanz des MDX_ENB_IN GPIOs in der MHS-Datei

Das Verhalten der Ports nach Außen wird durch die ersten Zeilen der Abbildungen 31 und 32 beschrieben. Hier werden durch das Schlüsselwort „DIR“ die Port-Modi als „out“ festgelegt sowie die Bitbreite der Ports durch das Schlüsselwort „VEC“ (wie Vector) definiert.

```
1  PORT LAMBDA_CTRL_OUT_GPIO_O_pin = LAMBDA_CTRL_OUT_GPIO_IO_O, DIR=O,VEC=[0:31]
   ..
2  BEGIN xps_gpio
3   PARAMETER INSTANCE = LAMBDA_CTRL_OUT      #Instanzbezeichnung
4   PARAMETER HW_VER = 2.00.a                 #Versionsnummer des IP-Cores
5   PARAMETER C_GPIO_WIDTH = 32               # Breite des GPIOs (Bit)
6   PARAMETER C_BASEADDR = 0x81440000         # Baseadresse
7   PARAMETER C_HIGHADDR = 0x8144ffff         # Endadresse
8   BUS_INTERFACE SPLB = plb_v46_0            # Bus Interface
9   PORT GPIO_IO_O = LAMBDA_CTRL_OUT_GPIO_IO_O# Output Port
10  PORT GPIO_IO_I = LAMBDA_CTRL_OUT_GPIO_IO_O# Input Port
11 END
```

Abbildung 32: Definition der Instanz des LAMBDA_CTRL_OUT GPIOs in der MHS-Datei

Die zwei weiteren GPIOs die „MDX_ENB_OUT“ und „LAMBDA_CTRL_IN“ sind als reine Inputports konfiguriert. Die „GPIO_DATA“-Register, „GPIO_TRI“-Reister sowie Tri-State Buffer werden nicht verwendet. Stattdessen beinhalten die GPIOs die „Interrupt Detection Logic“ welche durch die „IP2INTC_Irpt“ Ports die Interruptrequests an den Interrupt Controller des Systems weiter leitet. Das Interrupt Konzept des Systems wird nachfolgend erklärt. Die Definition der Instanzen der GPIOs sind in der Abbildungen 33 und 34 dargestellt.

```
1  PORT MDX_ENB_OUT_GPIO_I_pin = net_MDX_ENB_OUT_GPIO_I_pin, DIR=I, VEC=[0:11]
   ..
2  BEGIN xps_gpio
3   PARAMETER INSTANCE = MDX_ENB_OUT    #Instanzbezeichnung
4   PARAMETER HW_VER = 2.00.a           #Versionsnummer des IP-Cores
5   PARAMETER C_GPIO_WIDTH = 12         # Breite des GPIOs (Bit)
6   PARAMETER C_INTERRUPT_PRESENT = 1   #Interrupt Logik vorhanden
7   PARAMETER C_BASEADDR = 0x81400000   # Baseadresse
8   PARAMETER C_HIGHADDR = 0x8140ffff   # Endadresse
9   BUS_INTERFACE SPLB = plb_v46_0      # Bus Interface
10  PORT GPIO_IO_I = net_MDX_ENB_OUT_GPIO_I_pin # Input Port
11  PORT IP2INTC_Irpt = MDX_ENB_OUT_IP2INTC_Irpt# Interrupt Port
12 END
```

Abbildung 33: Definition der Instanz des MDX_ENB_OUT GPIOs in der MHS-Datei

```
1  PORT LAMBDA_CTRL_IN_GPIO_I_pin=net_LAMBDA_CTRL_IN_GPIO_I_pin,DIR=I,VEC=[0:31]
   ..
2  BEGIN xps_gpio
3   PARAMETER INSTANCE = LAMBDA_CTRL_IN#Instanzbezeichnung
4   PARAMETER HW_VER = 2.00.a           #Versionsnummer des IP-Cores
5   PARAMETER C_GPIO_WIDTH = 32         # Breite des GPIOs (Bit)
6   PARAMETER C_BASEADDR = 0x81460000   # Baseadresse
7   PARAMETER C_HIGHADDR = 0x8146ffff   # Endadresse
8   PARAMETER C_INTERRUPT_PRESENT = 1   #Interrupt Logik vorhanden
9   BUS_INTERFACE SPLB = plb_v46_0      # Bus Interface
10  PORT GPIO_IO_I = net_LAMBDA_CTRL_IN_GPIO_I_pin   # Input Port
11  PORT IP2INTC_Irpt = LAMBDA_CTRL_IN_IP2INTC_Irpt  # Interrupt Port
12 END
```

Abbildung 34: Definition der Instanz des LAMBDA_CTRL_IN GPIOs in der MHS-Datei

Die „Interrupt Detection Logic“ beinhaltet drei weitere Steuerungs - und Statusregister die durch PLB adressiert werden können. Die Bezeichnungen und Address-Offsets der Register sind in der Tabelle 6 zusammengefasst.

Register Name	Description	PLB Address	Access
GIER	Global Interrupt Enable Register	C_BASEADDR + 0x11C	Read/Write
IP IER	IP Interrupt Enable Register	C_BASEADDR + 0x128	Read/Write
IP ISR	IP Interrupt Status Register	C_BASEADDR + 0x120	Read/TOW[1]

1 Toggle-On-Write (TOW)

Tabelle 6: XPS GPIO Interrupt Register. Quelle [14] S.12

Das „Global Interrupt Enable Register“ (GIER) ist ein 32 Bit Register, welches das Interrupt Handlung des GPIOs steuert. Mit Bit 0 dieses Registers wird die Interrupt Generierung zugelassen oder gesperrt. Das „Interrupt Enable Register“ (IP IER) aktiviert die Interrupt Generierung jeweils für den ersten und zweiten Kanal des GPIOs. Das „Interrupt Status Register“ (IP ISR) beinhaltet zwei Flags die die Interruptrequests für jeden Kanal signalisieren.

5.1.1.3 XPS Serial Peripheral Interface

Das XPS SPI unterscheidet sich von den herkömmlichen seriellen Schnittstellen durch die optionalen FIFOs mit einer festen Länge von 16 Datenwörtern, welche zum Puffern der Daten vorgesehen sind sowie einer einstellbaren Breite des Schieberegisters, welches als 8, 16 oder 32 Bit konfiguriert werden kann. Das Blockschaltbild des SPI-Cores ist in der Abbildung 35 dargestellt.

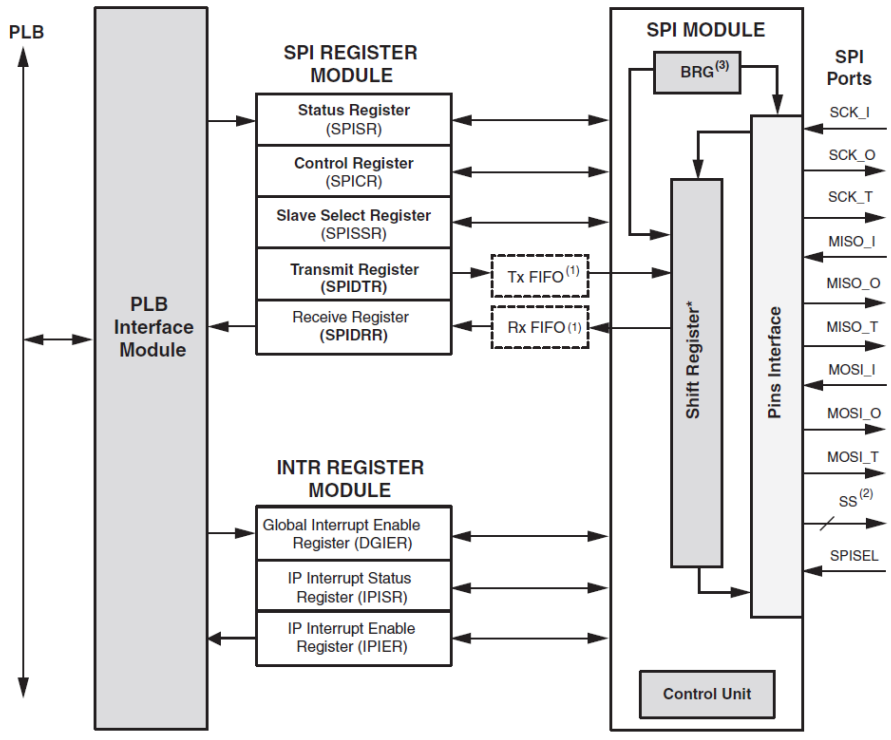


Abbildung 35: Blockschaltbild des XPS SPI. Quelle [15] S.2

In diesem Projekt ist das XPS SPI als Master implementiert und treibt durch die LVDS-Transmitter die Daten- und Clock Eingänge der Medipix3 Chips. Die Instantiierung des XPS SPI-Cores zeigt die Abbildung 36.

```
1  PORT PPC_SPI_SCK_O_pin = xps_spi_0_SCK_O, DIR = O
2  PORT PPC_SPI_MISO_I_pin = net_SPI_MISO_I_pin, DIR = I
3  PORT PPC_SPI_MOSI_O_pin = xps_spi_0_MOSI_O, DIR = O
   ..
4  BEGIN xps_spi
5   PARAMETER INSTANCE = xps_spi_0      #Instanzbezeichnung
6   PARAMETER HW_VER = 2.02.a           #Versionsnummer des IP-Cores
7   PARAMETER C_SCK_RATIO = 2           # Taktteiler
8   PARAMETER C_NUM_TRANSFER_BITS = 8   # Breite des Schiftregisters
9   PARAMETER C_BASEADDR = 0x83400000   # Baseadresse
10  PARAMETER C_HIGHADDR = 0x8340ffff   # Endadresse
11  PARAMETER C_FIFO_EXIST = 1          # FIFO existiert
12  BUS_INTERFACE SPLB = plb_v46_0      # Bus Interface
13  PORT SCK_O = xps_spi_0_SCK_O        # SPI Clock
14  PORT MISO_I = net_SPI_MISO_I_pin    # Master Input Slave Output
15  PORT MOSI_O = xps_spi_0_MOSI_O      # Master Output Slave Input
16 END
```

Abbildung 36: Definition der Instanz des XPS SPI in der MHS-Datei

Die Breite des Schieberegisters wurde auf 8 Bit eingestellt. Die aktuelle Implementierung des XPS SPI-Cores beinhaltet die beiden FIFOs, was durch den „C_FIFO_EXIST=1“ Parameter, entsprechend der Zeile 11 in der Abbildung 36, festgelegt wurde. Dadurch wird die Übertragungsbandbreite dieser Schnittstelle erhöht. Die Übertragungsfrequenz der seriellen Schnittstelle wurde auf den höchst möglichen Wert eingestellt. Dieser lässt sich folgendermaßen ermitteln:

$$\frac{SPLB\_CLK}{C\_SCK\_RATIO} = \frac{100MHz}{2} = 50MHz$$

Gleichung 10:Die serielle Übertragungsfrequenz des SPIs

Hier bezeichnet *SPLB_CLK* die Taktfrequenz des PLBs und beträgt in diesem Design 100 MHz, *C_SCK_RATIO* ist ein Generic Parameter welches entsprechend der Zeile 7 der Abbildung 36 initialisieret wurde.

In Abbildung 37 die der Quelle [15] S.23 entnommen wurde ist ein Zeitdiagramm eines Schreibzugriffes auf SPI dargestellt. Aus diesem Diagramm ist erkennbar, dass ein Schreibzugriff durch PLB vier Taktzyklen dauert. Bis das erste Bit am seriellen Datenausgang (MOSI) der Schnittstelle ausgegeben wird, werden weitere 6 Taktzyklen benötigt und die Übertragung von 8 Datenbits dauert weitere 32 Takte.

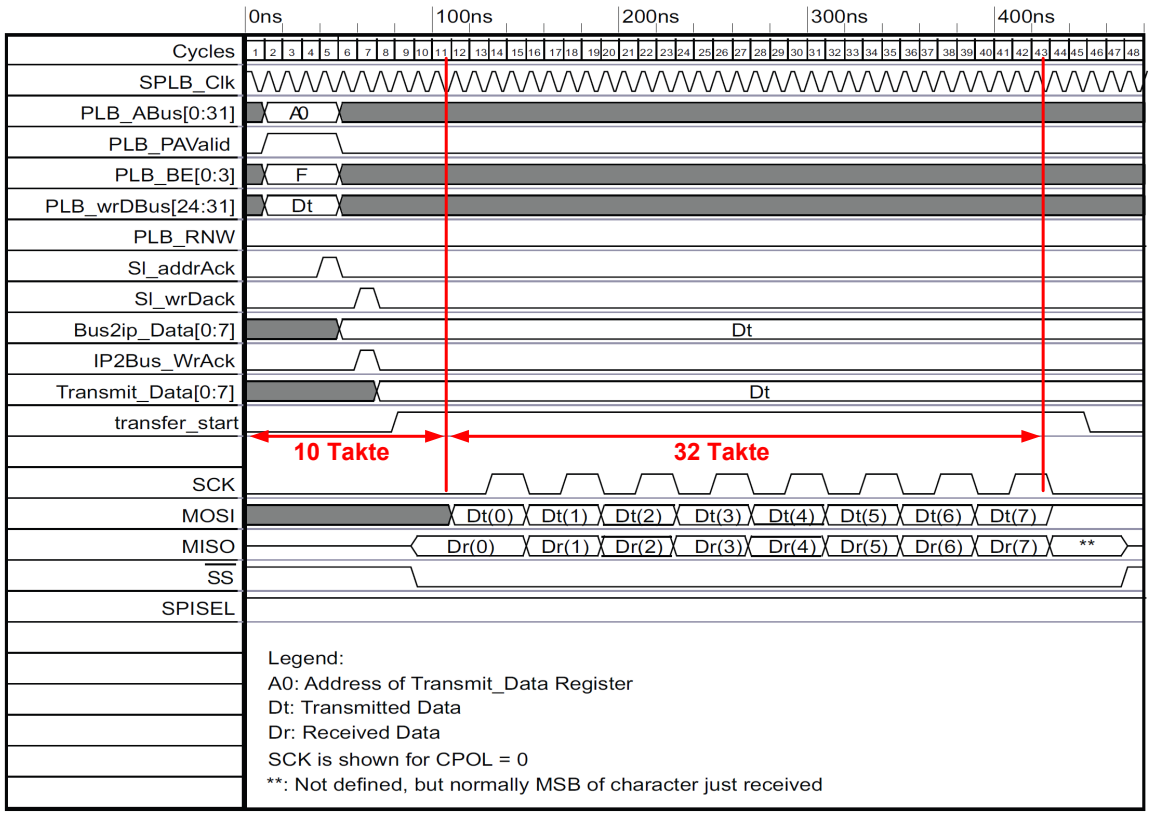


Abbildung 37: Zeitdiagramm eines Schreibzugriffes auf SPI. Quelle [15] S.23

Allerdings beträgt das Takt-Ratio hier „4“ wohingegen in der aktuellen Implementierung des SPIs einen Wert von „2“ erreicht wird. Deshalb werden für die Übertragung von 8 Datenbits in diesem Fall nur noch 16 Taktzyklen benötigt. Somit werden für die Übertragung eines Datenbytes insgesamt 26 Taktzyklen benötigt. Mit der Taktfrequenz des PLBs von 100 MHz benötigt diese Transaktion eine Übertragungszeit von:

$$\frac{1}{100 \times 10^{6}\,Hz} \times 26 = 260\,ns$$

Gleichung 11:Übertragungszeit eines Bytes durch das SPI

5.1.1.4 Interrupt Konzept des Systems

Abbildung 38 stellt das Interrupt Konzept des eingebetteten Systems dar. Die zentrale Einheit ist dabei der Interrupt Controller „xps_intc“, welcher die Aufgabe hat die vier Interruptquellen an einen Eingang des PowerPC440 Prozessors anzubinden.

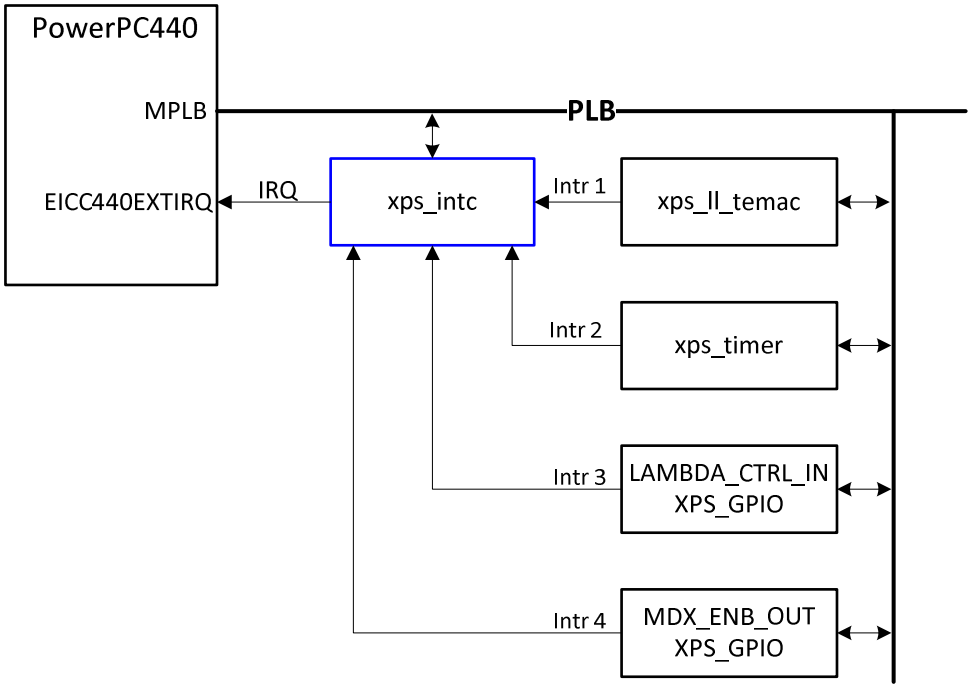


Abbildung 38: Interrupt Konzept des eingebetteten Systems

Die in Abbildung 38 dargestellten Interruptquellen haben folgende Funktionalität:

- **xps_ll_temac**: Dieser Interruptrequest wird erzeugt wenn ein Ethernet Frame empfangen wird.
- **xps_timer**: Erzeugt einen Interruptrequest beim Überlauf des Timers.
- **LAMBDA_CTRL_IN** (XPS_GPIO**)**: Erzeugt einen Interruptrequest bei der Änderung des Pegels am Externen Synchronisationseingang des Detektors.
- **MDX_ENB_OUT** (XPS_GPIO): Dieser Interruptrequest wird bei einer Pegeländerung an den „Enable_Out“- Ausgängen der Medipix3 Chips erzeugt.

Der PowerPC440 Prozessor verfügt insgesamt über zwei Eingänge für die externe Interrupts: einen „EICC440CRITIRQ“-Eingang für die kritischen Interrupts und einen zweiten „EICC440EXTIRQ“-Eingang für unkritischen Interrupts. Letzterer wird für die Annahme der Interruptrequests des Interrupt Controllers verwendet. Das Blockschaltbild des Interrupt Controllers stellt die Abbildung 39 dar.

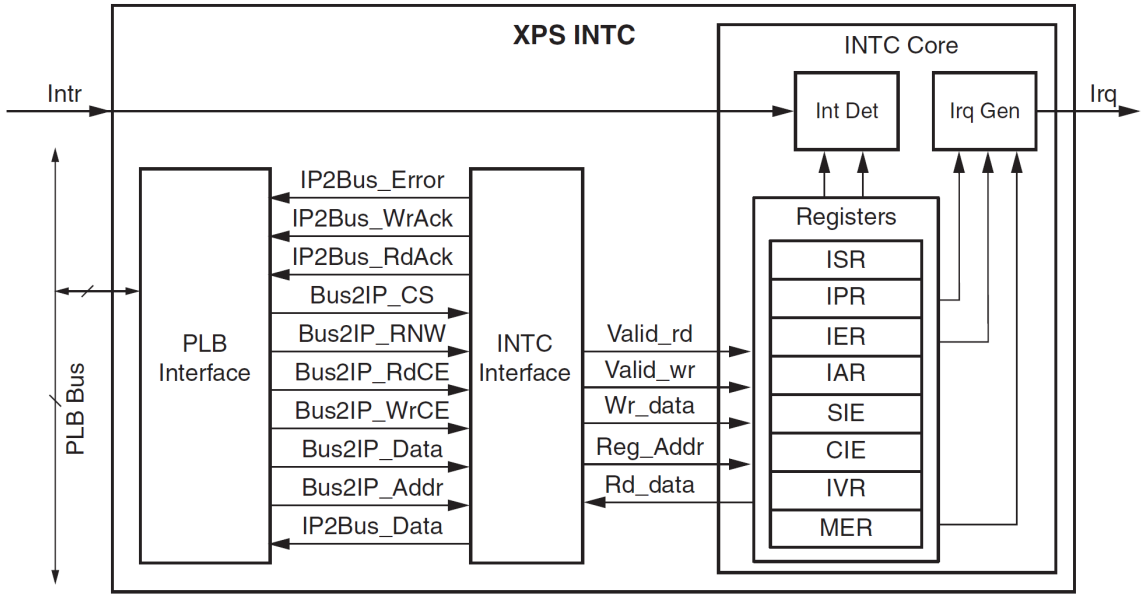


Abbildung 39: Blockschalbild des Interrupt Controllers. Quelle [21] S.3

Zum Verwalten der Interruptrequests der Peripheriekomponenten verfügt der Interrupt Controller über acht 32 Bit Register deren Bezeichnung und Address-Offsets in der Tabelle 7 zusammengefasst sind.

Register Name	Base Address + Offset (Hex)	Access Type	Abbreviation	Reset Value
Interrupt Status Register	C_BASEADDR + 0x0	Read / Write	ISR	All Zeros
Interrupt Pending Register	C_BASEADDR + 0x4	Read	IPR	All Zeros
Interrupt Enable Register	C_BASEADDR + 0x8	Read / Write	IER	All Zeros
Interrupt Acknowledge Register	C_BASEADDR + 0xC	Write	IAR	All Zeros
Set Interrupt Enable Bits	C_BASEADDR + 0x10	Write	SIE	All Zeros
Clear Interrupt Enable Bits	C_BASEADDR + 0x14	Write	CIE	All Zeros
Interrupt Vector Register	C_BASEADDR + 0x18	Read	IVR	All Ones
Master Enable Register	C_BASEADDR + 0x1C	Read / Write	MER	All Zeros

Tabelle 7: Register des Interrupt Controllers. Quelle [21] S.10

Jedem Bit eines Registers kann jeweils eine Interruptquelle zugeordnet werden, wobei die niedrigere Bitposition innerhalb eines Registers die höhere Priorität des Interrupts bedeutet. In dem aktuellen Systemdesign werden entsprechen der Abbildung 38 vier Quellen durch den Interrupt Controller verwaltet, wobei die Quelle „xps_ll_temac“ die höchste- und „MDX_ENB_OUT“ die niedrigste Priorität haben. Die Anbindung der Interruptquellen an den Interrupt Controller erfolgt durch die Verkettung der Inerruptsignale zu einem Bitvektor, welcher an den Port des Controllers angeschlossen wird. Die Verkettung der Interruptsignale ist in Zeile 8 der Abbildung 40 gezeigt.

```
1  BEGIN xps_intc
2   PARAMETER INSTANCE = xps_intc_0     #Instanzbezeichnung
3   PARAMETER HW_VER = 2.01.a           #Versionsnummer des IP-Cores
4   PARAMETER C_BASEADDR = 0x81800000   # Baseadresse
```

```
5   PARAMETER C_HIGHADDR = 0x8180ffff   # Endadresse
6   PARAMETER C_IRQ_IS_LEVEL = 1
7   BUS_INTERFACE SPLB = plb_v46_0      # Bus Interface
8   PORT Intr = Hard_Ethernet_MAC_TemacIntc0_Irpt & xps_timer_0_Interrupt &
                LAMBDA_CTRL_IN_IP2INTC_Irpt & MDX_ENB_OUT_IP2INTC_Irpt
10  PORT Irq = ppc440_0_EICC440EXTIRQ   # Interruptrequest
11 END
```

Abbildung 40: Definition der Instanz des Inerrupt Controllers in der MHS Datei

Sollte eine oder mehreren Quellen einen Interrupt Request signalisieren, wird dies durch die Interrupt Detection Logik (Abbildung 39) registriert und die entsprechenden Bits des Interrupt Status Register (ISR) werden gesetzt. Daraufhin wird durch den Interrupt Request Generator (Abbildung 39) einen Interrupt Request für den PowerPC440 Prozessor erzeugt. Danach erfolgt einen Zugriff des Prozessors mittels PLB auf das Interrupt Vector Register (IVR), welches die Nummer der aktiven Interruptquelle mit der höchsten Priorität enthält. Dadurch ermittelt der Prozessor welche Interruptquelle den Interrupt ausgelöst hat. Nach der Abarbeitung der jeweiligen Interrupt Service Routine wird der Interruptrequest durch das Setzen des einsprechenden Bits im Inerrupt Acknowlege Register (IAR) gelöscht.

5.1.2 Entwurf eines Lokallink basierten DMA - Interfaces

Das auf dem LokalLink basierte DMA Interface „Lambda_LL_Interface“ wurde als eigener IP-Core entwickelt welcher sich im „pcores“ Verzeichnis des XPS Projektes befindet. Der IP-Core verfügt über folgende Schnittstellen: eine PLB Schnittstelle zum Anschließen des IP-Cores an den PowerPC440 Prozessor als Slave, eine LocalLink Schnittstelle zum Anbinden des IP-Cores an den DMA Controller des eingebetteten Prozessor Blocks und eine Schnittstelle zum Anschließen des Parallel Data Readouts an das IP-Core. Für die Entwicklung dieses IP-Cores wurde der von Xilinx XPS zur Verfügung gestellte „Create or Import Peripheral“ Wizard verwendet. Mit Hilfe dieses Tools wird ein VHDL Template für das PLB Slave Device erstellt, welches über einen PLBv4.6 Slave IP Interface und einen User Logic Modul verfügt. Eine ausführliche Beschreibung des Entwurfsprozesses eines VHDL Templates kann dem Kapitel 8 der Quelle [22] entnommen werden.

In Abbildung 41 ist das Blockschaltbild des IP-Cores nach der Ergänzung des User Logic Moduls mit eigenem VHDL Code dargestellt.

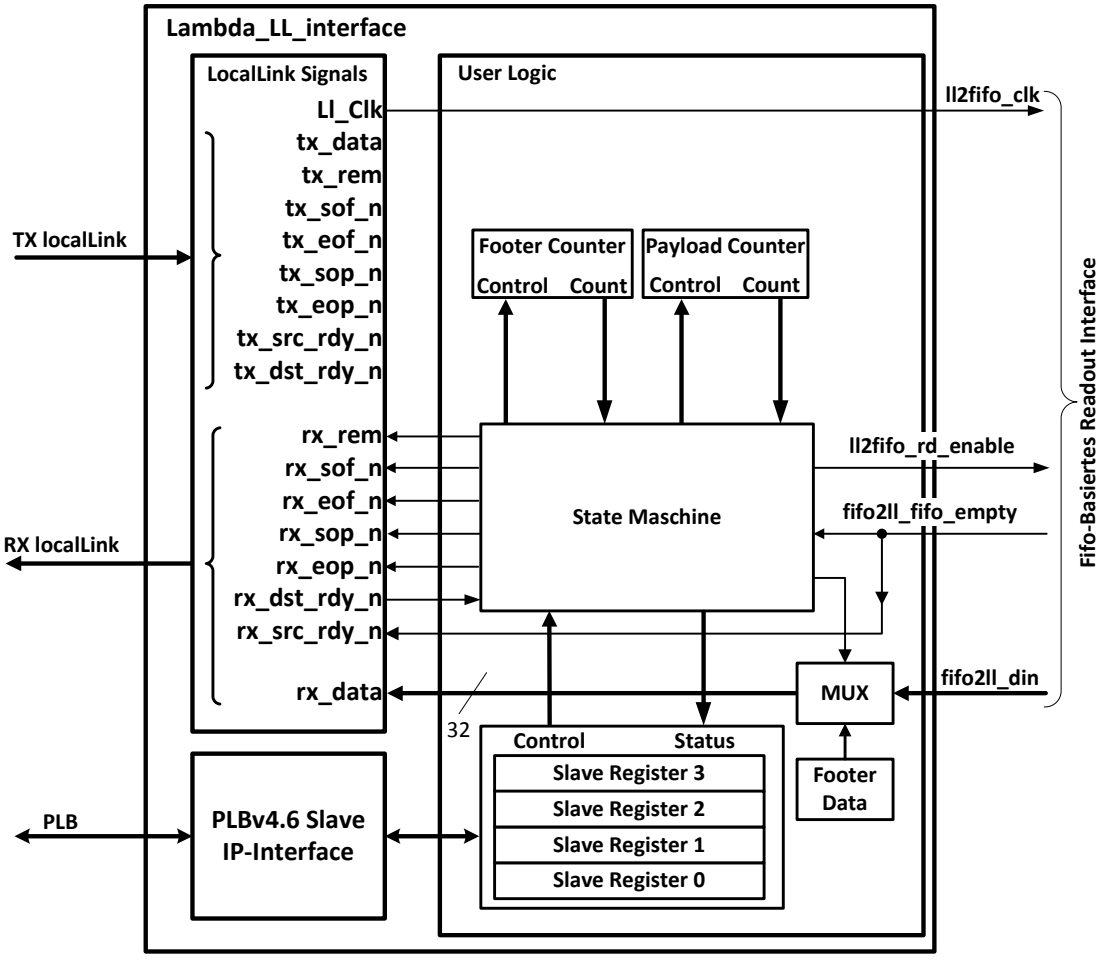


Abbildung 41: Blockschaltbild eines LocalLink-Basierten Interfaces „Lambda_LL_Interface“

Zum Austauschen der Control- und Status-Information mit dem IP-Core wurden vier 32 Bit Slave Register implementiert, welche entsprechend der Tabelle 8 durch PLB adressiert werden können.

Register Bezeichnung	Base Address + Offset (Hex)	Zugriffsart
Slave Register 0	C_BASEADDR+0x00	Lesen/Schreiben
Slave Register 1	C_BASEADDR+0x04	Lesen/Schreiben
Slave Register 2	C_BASEADDR+0x08	Lesen/Schreiben
Slave Register 3	C_BASEADDR+0x0C	Lesen/Schreiben

Tabelle 8: Address MAP der Slave Register des IP-Cores

Die Anbindung des Parallel-Readouts an den IP-Core erfolgt mit Hilfe eines FIFOs[34], wobei das FIFO ein Bestandteil des Parallel-Readouts ist. Durch die Verwendung des FIFOs als Verbindungsglied wurden zwei unterschiedliche Clock Domänen realisiert, die das asynchrone Schreiben und Lesen der Daten in und aus dem FIFO ermöglichen. Dies ist notwendig weil der LocalLink und der Parallel-Readout unterschiedlich getaktet werden. Das LocalLink Interface und der „Lamnbda_ll_interface“ IP-Core werden mit 200MHz getaktet. Der Parallel Radout und die Medipix3 Chips werden dagegen mit einem 100MHz Takt betrieben. Die Anbindung des Parallel-Readouts an den LocalLink stellt die Abbildung 42 dar.

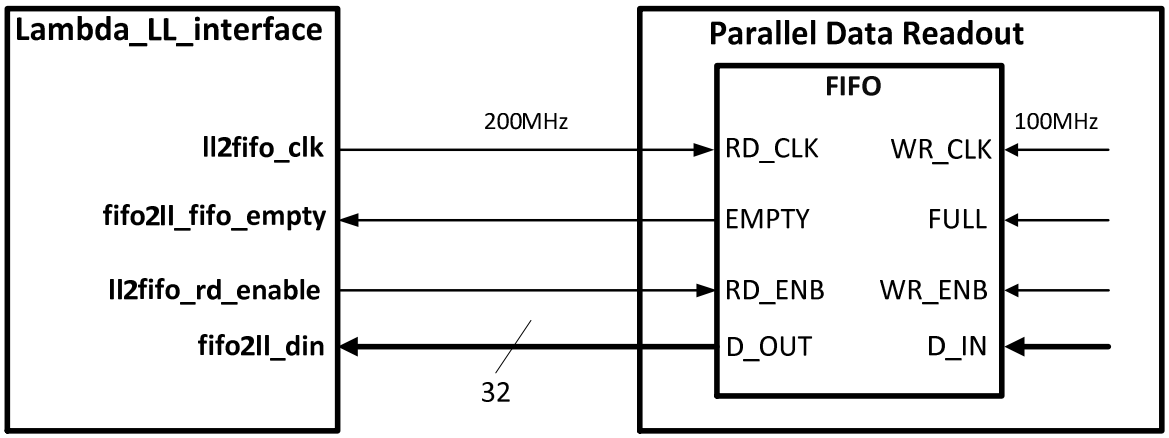


Abbildung 42: Anbindung des Parallel-Readouts an das Lambda_LL_interface

Zum Lesen der Daten aus dem FIFO sind folgende I/Os implementiert:

- fifo2ll_din: 32 Bit Eingangsport des Lambda_LL_interfaces, für die Übertragung der Matrixdaten.
- ll2fifo_rd_enable: aktiviert das Lesen der Daten aus dem FIFO.
- fifo2ll_fifo_empty: Eingangsport zum Anschließen des Empty-Flags des FIFOs, wird gesetzt wenn der FIFO ist leer.
- ll2fifo_clk: LocalLink Takt zum Lesen der Daten aus dem FIFO.

[34] First In First Out. Quelle[5, S.139]

Das LocalLink Interface des DMA-Controllers ist bidirektional und verfügt über jeweils einen Datenübertragungskanal in jede Übertragungsrichtung. Für die Lesezugriffe auf das Memory und die Übertragung der Daten von dem Memory Controller in Richtung Peripherie steht ein Tx-Kanal zur Verfügung. Für die Übertragung der Daten von der Peripherie und die Schreibzugriffe auf das Memory wird einen RX-Kanal verwendet. In der aktuellen Implementierung des Interfaces kommt nur der Rx-Kanal zur Anwendung, da die Matrixdaten der Medipix3 Chips, vom Parallel-Readout in das verwendete DDR2-SDRAM geschrieben werden sollen. Die Signale des Tx-Kanals wurden in der Top-Entity des IP-Cores nur der Vollständigkeit halber aufgenommen. Somit lassen sich die beiden Kanäle zu einem Bus-Interface zusammenfassen. Dies erleichtert das Anbinden des IP-Cores an das LoclLink Interface des eingebetteten Prozessor Blocks im XPS.

Die Übertragung der Daten über den LocalLink erfolgt paketorientiert unter Verwendung eines LocalLink Protokolls. Das Format eines Datenpaketes stellt die Abbildung 43 dar. Das Paket besteht aus einem Header, einem Payload und einem Footer. Der Payload Bereich ist für die Übertragung der Nutzdaten vorgesehen. Dessen Länge kann beliebig gewählt werden. In der aktuellen Implementierung beträgt der Payload Bereich 1180032 Byte, was der Größe eines Lambda Images entspricht. Der Footer ist 8 Words lang und wird für die Flusssteuerung der Datenübertragung von der DMA-Engine verwendet.

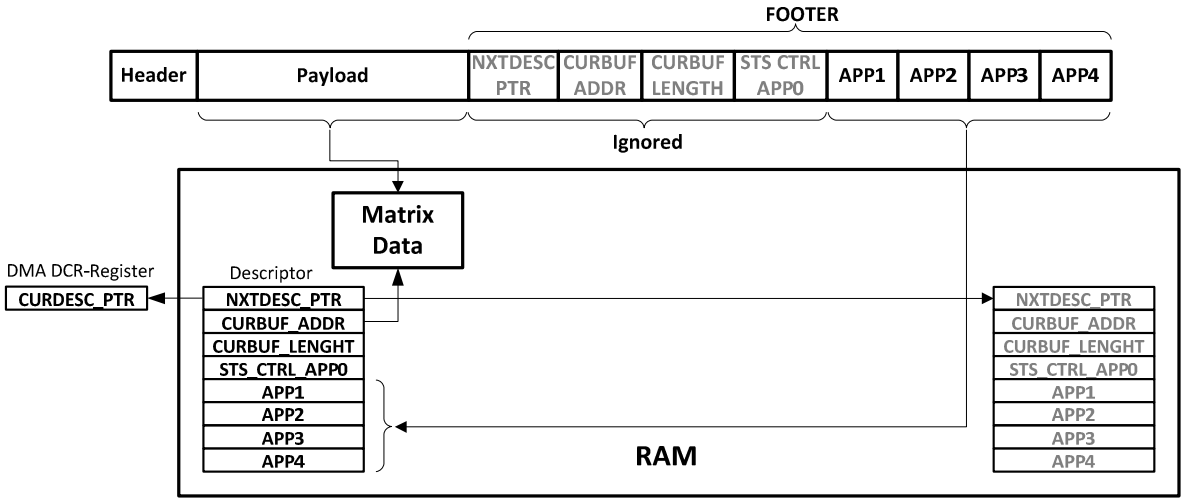


Abbildung 43: Format eines LocalLink Paketes. Nach Quelle [3] S.234

Die Header Information für die Datenübertragung wird nicht benötigt und wird daher von der DMA-Engine verworfen. Die Lokalisierung der Payload Daten im Speicher erfolgt mit Hilfe eines Deskriptors. Dafür soll die Adresse des im Speicher lokalisierten Descriptors in das DCR-Register „CURDESC_PTR“ der DMA-Engine vor der Übertragung des Datenpaktes vom Prozessor geschrieben werden. Das Format des Deckriptors kann der Seite 229 der Quelle [3] entnommen werden.

Für die Kenzeichnung des Headers, des Payloads und des Footers von einem Datenpaket werden die jeweiligen Flags des Interfaces verwendet. Der Anfang des Headers wird durch den Low-Pegel des „Start of Frame“ Flags „rx_sof_n“ signalisiert. Die „Start of Payload“ Flag „rx_sop_n“ kennzeichnet den Anfang des Payload Bereiches und die „End of

Payload“ Flag „rx_eof_n“ signalisiert dem entsprechend das Ende des Payload Bereiches und gleichzeitig den Anfang des Packet-Footers. Schließlich wird das Ende des Paketes durch den „End of Frame“ Flag „rx_eof_n“ signalisiert. Das Zeitdiagramm des LocalLink Protokolls die der Seite 235 der Quelle [3] entnommen wurde ist in der Abbildung 44 dargestellt.

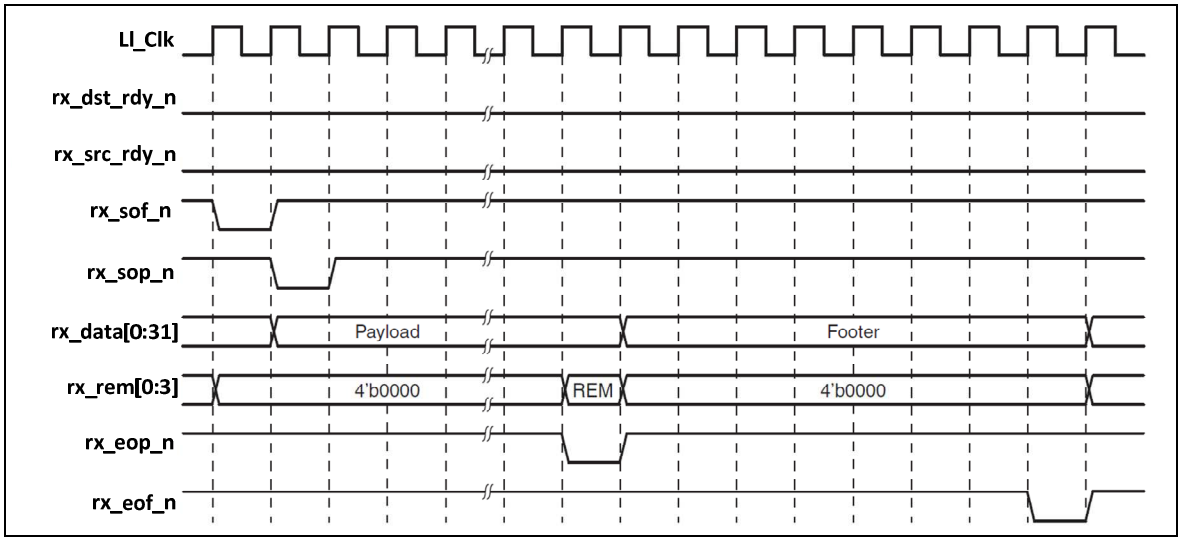


Abbildung 44: Zeitdiagramm des LocalLink Protokolls. Quelle [3] S.235

Zwei weiteren sehr wichtigen Flags des LocalLink Interfaces sind die „rx_src_rdy_n“ und „rx_dst_rdy_n“ welche die Bereitschaft für die Datenübertragung von Source und Destination signalisieren. Dabei wird in der aktuellen Implementierung das FIFO als Source bezeichnet und das LacalLink Interface des eingebetteten Prozessor Blocks, als Destination. Diese beiden Flags sind aktiv-low, sollten einer der beiden Flags in den Pegel-High wechseln, so muss die Datenübertragung angehalten werden bis diese beiden Flags wieder in den Zustand Low wechseln.

Für die Umsetzung des LocalLink Protokolls wird ein Zustandsautomat und zwei weitere Binärzähler für die Payload- und Footer Daten eingesetzt. Der Zustandautomat wurde als Moore-Automat entworfen. Dieser ist ein synchrones Schaltwerk bei dem die Ausgangssignale nur eine Funktion des aktuellen Automatenzustandes Z sind und der mit der nächsten positiven Taktflanke Folgezustand Z^+ aus dem aktuellen Zustand Z und die an dem Automaten anliegenden Eingangsignale vorausberechnet. Das Verhalten des Automaten kann durch folgende Gleichungen beschrieben werden:

$$Z^+ = F(rx\_dst\_rdy\_n, FIFO\_Empty, Payload\_Count, Footer\_Count, Slave\_\mathrm{Re}g, Z)$$

Gleichung 12

$$LL\_Flags = F(Z)$$

Gleichung 13

$$FIFO\_Rd\_enable = F(Z)$$

Gleichung 14

Hier wurden die „rx_sof_n“, „rx_sop_n“, „rx_eop_n“ und „rx_eof_n“ Signale unter „LL_Flags“ zusammengefasst.

Der Zustandsautomat wurde nach Huffman-Normalform beschrieben und beinhaltet zwei Prozesse: ein getakteter Prozess als Zustandsspeicher und ein kombinatorisches Prozess zur Bestimmung der Folgezustände und zur Aktualisierung der LocalLink Flags sowie des FIFO-Read-Enable Signals. Die Funktionsübersicht des Automaten stellt die Abbildung 45 dar.

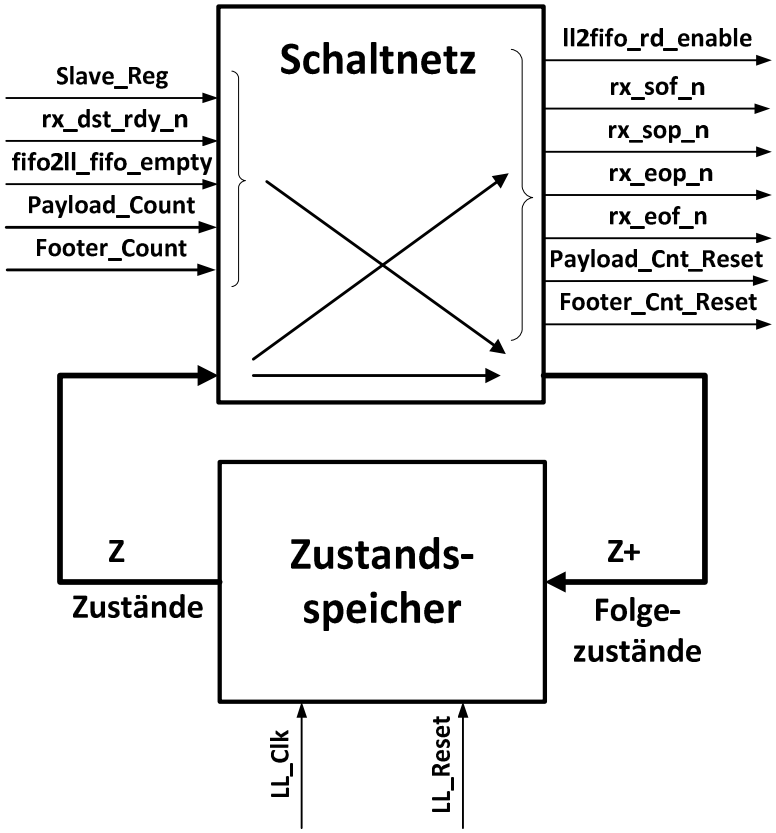


Abbildung 45: Funktionsübersicht des Zustandsautomaten

Das Zustandsdiagramm des Automaten ist in der Abbildung 46 dargestellt und beinhaltet folgende Zustände:

- **IDLE:** Ist ein Anfangszustand in dem sich der Automat nach der Initialisierung befindet. Der Automat geht in den nächsten „RX_SOF“ Zustand über wenn das LocalLink zum empfangen der Daten bereit ist, der FIFO nicht leer ist und die Datenübertragung durch das LSB[35] des Slave_Reg0 zugelassen wird. Das Slave Register 0 wird durch den PLB vom Prozessor aus initialisiert.

- **RX_SOF:** In dem „Start of Frame“ Zustand erfolgt für eine Taktperiode die Zuweisung eines Low-Pegels an das rx_sof_n Signal. Der Übergang in den nächsten

[35] Low Signifikant Bit

„RX_SOP“ Zustand erfolgt wenn Source (FIFO) und Distination (LocalLink) bereit sind. Sollten Source oder Distination keine Bereitschaft signalisieren kehr der Zustandautomat wieder in den IDLE Zustand zurück.

- **RX_SOP:** Im „Start of Payload“ Zustand wird das „rx_sop_n“ Signal für eine Taktperiode auf „Low“ geschaltet und gleichzeitig das „ll2fifo_rd_enable“ Signal auf High-Pegel gesetzt. Dadurch wird die Übertragung der Daten gestartet. Des Weiteren wird der Payload Counter aktiviert und gestartet. Unmittelbar danach geht der Automat in den nächsten „WRITE_DATA“ Zustand über. Der Payload Counter Zählt dabei nebenläufig die Menge der übertragenen Daten.

- **WRITE_DATA:** In diesem Zustand befindet sich der Automat während der gesamten Übertragung eines Lambda Images. Der laufende Zahlerstand des Payload Counter wird mit dem Inhalt des Slave Register 1 verglichen. Der Automat wachselt in den „FOOTER“ Zustand wenn der Zählerstand des Payload Counter einen Wert von 1180032 erreicht hat welches der Länge eines Lambda Images entspricht. Beim Übergang in den nächsten Zustand wird dem „rx_eop_n“ Flag ein Low-Pegel für eine Taktperiode zugewiesen.

- **FOOTER:** Der „FOOTER“ Zustand dient der Übertragung der acht Footer-Worte. Dafür wird der Footer Counter aktiviert und gestartet. Beträgt der Zählerstand des Footer Counters einen Wert von „0x06“ geht der Automat in den nächsten Zustand „RX_EOF“ über.

- **RX_EOF:** In diesem „End of Frame“ Zustand erfolgt für eine Taktperiode die Zuweisung des Low-Pegels des „rx_eof_n“ Signals. Des Weiteren wird der Footer Counter gestoppt und der Automat wechselt in den letzten „WAIT_FOR_END“ zustand.

- **WAIT_FOR_END:** In diesem Zustand befindet sich der Automat solange das Slave Register 0 einen Wert von „0x01“ aufweist. Durch das initialisieren des Slave Registers 0 mit „0x00“ kehrt der Automat in den IDLE Zustand zurück. Dieser Zustand wurde zum Verhindern ungewünschter Datenübertragungsversuche eingeführt und ermöglicht dem PowerPC 440 Prozessor der Übertragungsstartpunkt festzulegen.

Der vollständige VHDL-Code des Zustandsautomaten wird im Anhang der Anlage 1 aufgelistet. Der beim Erstellen des IP-Cores automatisch generierter Code des Slave PLB-Interfaces ist in dieser Auflistung nicht enthalten.

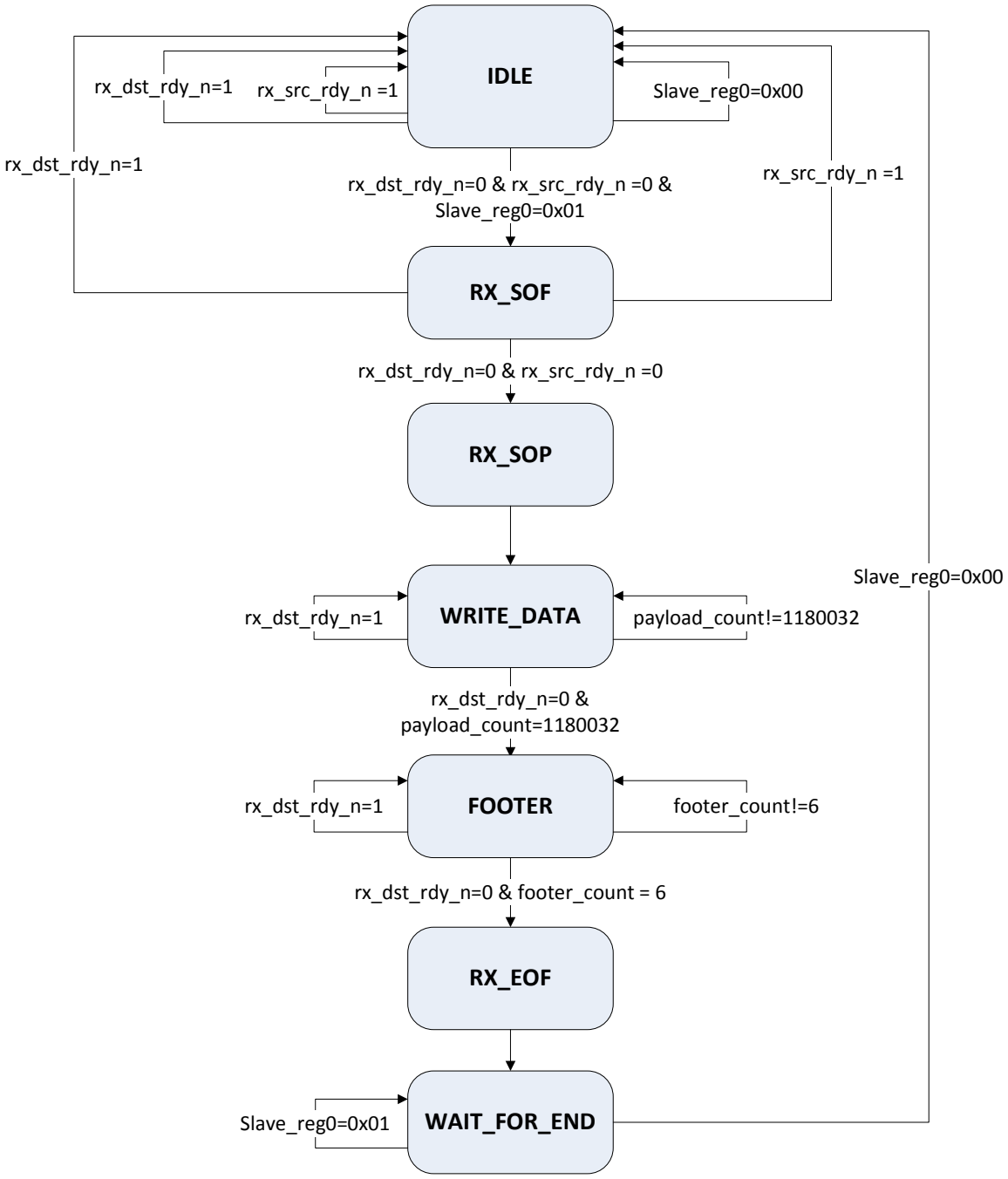


Abbildung 46: Zustandsdiagramm des Zustandsautomaten

Nach der Fertigstellung des IP-Core Designs wird dieses mit Hilfe des „Create or Import Peripheral“ Wizards in das XPS Projekt importiert. Dadurch werden die LocalLink Ports der Top-Entity des IP-Cores dem XSP bekannt gegeben. Dies gescheit durch das Editieren der MPD[36] Datei des IP-Cores durch den „Create or Import Peripheral“ Wizard.

Um die Implementierung des IP-Cores in das XPS Projekt zu vereinfachen, wurden alle LocalLink Ports bis auf „LL_Clk“ zu einem Bus zusammengefasst. Dafür wurde der Local-Link-Bus als Bus-Interfase entsprechend der Zeile 2 der Abbildung 47 in der MPD-Datei

[36] Microprocessor Peripheral Definition. Quelle [20, S.27]

deklariert und alle Ports des LocalLink Interfaces als Bus-Signale durch das Atribut „BUS = LLINK0“ definiert.

```
1  ## Bus Interfaces
2  BUS_INTERFACE BUS = LLINK0, BUS_STD = XIL_LL_DMA, BUS_TYPE = INITIATOR

3  ## Ports
4  PORT Ll_Clk = "", DIR = I, SIGIS = CLK
5  PORT Ll_Rst = LL_RST_ACK, DIR = I, BUS = LLINK0

6  # Initiator
7  PORT rx_data = LL_Rx_Data, DIR = O, VEC = [31:0], BUS = LLINK0
8  PORT rx_rem = LL_Rx_Rem, DIR = O, VEC = [3:0], BUS = LLINK0
9  PORT rx_sof_n = LL_Rx_SOF_n, DIR = O, BUS = LLINK0
10 PORT rx_eof_n = LL_Rx_EOF_n, DIR = O, BUS = LLINK0
11 PORT rx_sop_n = LL_Rx_SOP_n, DIR = O, BUS = LLINK0
12 PORT rx_eop_n = LL_Rx_EOP_n, DIR = O, BUS = LLINK0
13 PORT rx_src_rdy_n = LL_Rx_SrcRdy_n, DIR = O, BUS = LLINK0
14 PORT rx_dst_rdy_n = LL_Rx_DstRdy_n, DIR = I, BUS = LLINK0

15 # Target
16 PORT tx_data = LL_Tx_Data, DIR = I, VEC = [31:0], BUS = LLINK0
17 PORT tx_rem = LL_Tx_Rem, DIR = I, VEC = [3:0], BUS = LLINK0
18 PORT tx_sof_n = LL_Tx_SOF_n, DIR = I, BUS = LLINK0
19 PORT tx_eof_n = LL_Tx_EOF_n, DIR = I, BUS = LLINK0
20 PORT tx_sop_n = LL_Tx_SOP_n, DIR = I, BUS = LLINK0
21 PORT tx_eop_n = LL_Tx_EOP_n, DIR = I, BUS = LLINK0
22 PORT tx_src_rdy_n = LL_Tx_SrcRdy_n, DIR = I, BUS = LLINK0
23 PORT tx_dst_rdy_n = LL_Tx_DstRdy_n, DIR = O, BUS = LLINK0
```

Abbildung 47: Definition des LocalLink Buses in der MPD-Datei

Abbildung 48 zeigt einen Ausschnitt der XPS GUI in dem das LocalLink Interface des „Lambda_ll_dma“ IP-Cores als Bus vom XPS erkannt wird.

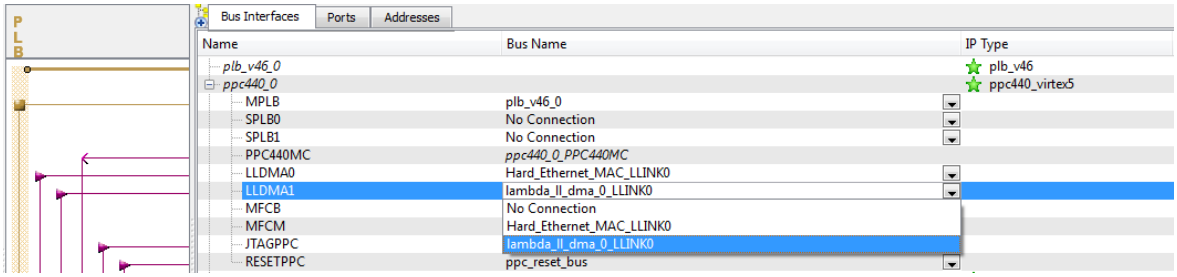


Abbildung 48: Inplementierung des LacalLink Buses im XPS

5.1.3 Taktung des LAMBDA-Moduls und der Readout Komponenten

Zum Lesen der Pixel-Matrix mit unterschiedlichen Geschwindigkeiten werden die Medipix3 Chips und Readout-Komponenten unterschiedlich getaktet. Wie bereits beschrieben, sieht das Detektorkonzept zum Übertragen der Daten zwei Ethernet Interfaces mit 1Gbit/s und 10Gbit/s vor. Beim 1Gbit Interface werden die Matrixdaten des LAMBDA-Moduls durch den DMA-Kanal des eingebeteten Prozessorblocks in das, dem Prozessor zugeordnete DDR2-Memory, zwischengespeichert und anschließend zum PC übertragen. Der DMA-Kanal des eingebetteten Prozessor-Blocks ist 32bit-breit und wird mit 200MHz getaktet. Daraus resultiert eine theoretische, maximale Übertragungsbandbreite von:

$$200 \times 10^6 Hz \times 32bit = 800MByte/s$$

Allerdings kann, laut Xilinx Dokumentation. [6] S.233, der Datenstrom des DMA-Kanals beliebig oft unterbrochen werden. Dadurch hängt die tatsächliche effektive Übertragungsbandbreite von verschiedenen Faktoren ab und liegt mit ca. 700MBytes/s deutlich unter der maximalen Übertragungsrate. Um Daten kontinuierlich ohne Verlust übertragen zu können wir das LAMBDA-Modul deshalb mit einem Taktsignal von 50 MHz getaktet wodurch ein Datenstrom von 600 MByte/s erzeugt wird.

Für Anwendungen bei denen deutlich höhere Framraten notwendig sind, wird zum Übertragung der Daten das 10G-Interface verwendet. So steht eine größere Übertragungsbandbreite zur Verfügung und dementsprechend kann das LAMBDA-Modul und dessen Readout-Komponente schneller getaktet werden. In diesem Fall wird ein 100MHz Takt verwendet, wodurch sich die Menge der durch das LAMBDA-Modul erzeugten Daten auf 1.2 GByte/s verdoppelt.

Die Ansteuerung der Medipix3 Chips erfolgt wie bereits beschrieben durch das SPI-Interface. Deshalb muss zum Übertragen der Steuersequenzen der SPI-Takt verwendet werden. Da der SPI_Takt Burstweise erfolgt und somit langsam ist, muss nach der Übertragung der jeweiligen Steuersequenz auf ein kontinuierliches Taktsignal umgeschaltet werden.

Die benötigten 50 MHz und 100 MHz Taktsignale werden durch einen „Digital Clock Manager“ [4] S.93 innerhalb des FPGAs erzeugt. Zum Umschalten der Taktsignale werden zwei Takt-Multiplexer eingesetzt. Das Blockschaltbild des Taktungsszenarios wird in der Abbildung 49 dargestellt. Die Implementierung der Komponenten wird nachfolgend beschrieben.

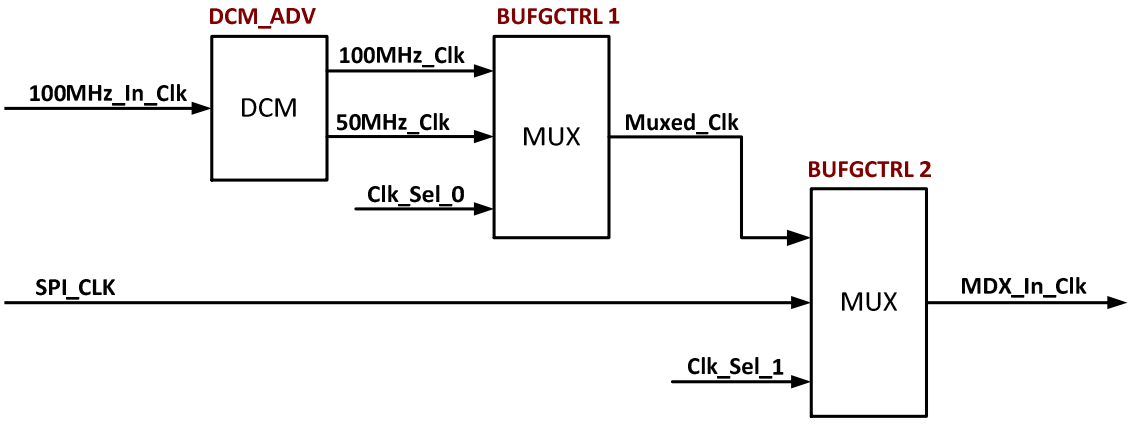


Abbildung 49: Taktung Szenario des LAMBDA-Modules und Readou-Komponenten

5.1.3.1 Digital Clock Manager

Der Xilinx XC5VFX70T FPGA verfügt über 12 DCM[37]-Blöcke. Diese befinden sich in sogenannten Clock Management Tiles (CMT). Jedes CMT verfügt über eine Phase Locked Loop (PLL) Komponente und zwei DCM-Blöcke. [5] S.47 Mit DCM-Blöcken lassen sich aus einem Referenztakt, mehrere Taktsignale mit so wohl höherer, als auch niedrigerer Frequenz sowie variabler Phasenlage generieren. Die DCM-Blöcke eines Virtex5 FPGAs können über zwei Library-Makros dynamisch konfigurierbar (DCM_ADV) bzw. in statischer Konfiguration (DCM_BASE) instantiiert werden. [5] S.47 Bei der Instantiierung des DCMs durch das DCM_ADV-Makro sind im Gegensatz zum DCM_BASE, die Phasenlagen der generierten Taktsignale nach der Implementierung im FPGA einstellbar und daher dynamisch konfigurierbar.

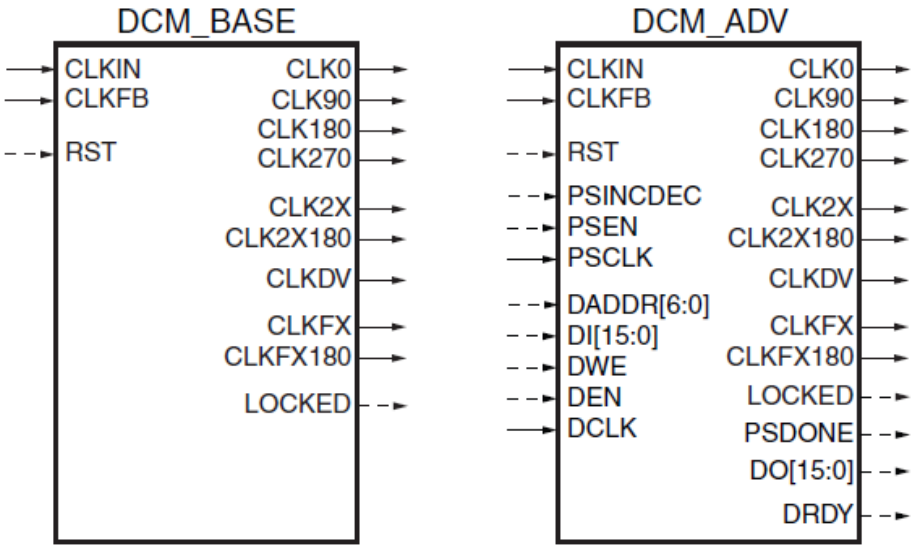


Abbildung 50: DCM_ADV und DCM_BASE Instanzen des DCMs. Quelle [5] S.59

Eingangsports:

- **CLKIN:** Takt-Referenzeingang. Aus diesem Taktsignal werden alle anderen Taktsignale generiert.
- **CLKFB:** Eingangsport für die Rückkopplung (FeedBack). Dieser Port dient der Phasensynchronisation der von DCM erzeugten Taktsignale mit dem Eingangstaktsignal des DCMs.

Ausgangsports:

- **CLK0:** Liefert einen Taktsignal mit der Frequenz und Phasenlage des Referenzsignals am Port CLKIN.

[37] Digital Clock Manager

- **CLK90:** Liefert einen Taktsignal mit Frequenz des CLK0-Signals und 90° Phasenverschiebung relativ zum CLK0.
- **CLK180:** Liefert ein Taktsignal mit Frequenz des CLK0-Signals und 180° Phasenverschiebung relativ zum CLK0.
- **CLK270:** Liefert ein Taktsignal mit Frequenz des CLK0-Signals und 270° Phasenverschiebung relativ zum CLK0.
- **CLK2X:** Liefert einen Taktsignal mit doppelter Frequenz und gleichen Phasenlage wie CLK0.
- **CLK2X180:** Taktsignal mit Frequenz des CLK2X-Signals und 180° Phasenverschiebung relative zum CLK2X.
- **CLKFX:** „Frequency-Synthesiss“ - Ausgangsport. Die Frequenz des Taktsignal an diesem Port ist einstellbar.. Die Ausgangsfrequenz ergibt sich aus folgender Gleichung:

$$CLKX\,(Freguenz) = (M/D) \times CLKIN\,(Frequenz)$$

  Wobei der Multiplikator „M“ ein „CLKFX_MULTIPLY“- Attribut im Wertebereich (2…32) repräsentiert und der Divisor „D“ repräsentiert ein „CLKF_DIVIDE“- Attribut im Wertebereich (1,5…16). [5] S.61

- **CLKFX180:** Invertierter CLKFX-Signal.

Die Digital Clock Manager der Virtex5 FPGA-Familie ermöglichen es, Taktsignale im Bereich von 32 MHz bis 550 MHz je nach FPGA-Typ zu erzeugen.

5.1.3.2 Erzeugung der Taktsignale

Zum Erzeugen 50 MHz und 100 MHz Frequenzen wird ein Digital Clock Manager (DCM) eingesetzt. Hierfür wird die DCM-Komponente durch DCM_ADV-Makro wie folgt instanziiert:

```
1     DCM_ADV_INST: DCM_ADV
2     generic map (
3           CLKFX_DIVIDE   => 4,
4           CLKFX_MULTIPLY => 2, -- 2/4 = 0,5
5           CLK_FEEDBACK   => "1X",
6           SIM_DEVICE     => "VIRTEX5"
7     )
8     PORT MAP (
9           CLKFB  => CLK0_OUT, -- 100MHz FeedBack
10          CLKIN  => CLKIN_IN, -- 100MHz Referenztakt
11          RST    => '0',
12          CLK0   => CLK0_BUF,
13          CLKFX  => CLKFX_BUF
14    );
15
16    CLK0_BUFG_INST : BUFG
```

```
17      PORT MAP (I => CLK0_BUF,
18                O => CLK0_OUT  -- 100MHz Taktausgang
19    );
20
21    CLKFX_BUFG_INST : BUFG
22      PORT MAP (I => CLKFX_BUF,
23                O => CLKFX_OUT -- 50MHz Taktausgang
24    );
```

Abbildung 51: Instantiierung der DCM-Komponente in der ISE-Umgebung

Als Referenz erhält das DCM ein Taktsignal mit Frequenz von 100 MHz am CLKIN-Port. Aus diesem Signal werden beide Ausgangssignale generiert. Der vom DCM am CLK0-Port generierter Taktsignal ist dem Referenztakt Frequenz- und Phasengleich. Dieses Signal wird mittels eines globalen Buffers mit der Taktmatrix des FPGAs verbunden und anschließend mit einem Eingang des Taktmultiplexers verdrahtet. Des Weiteren wird dieses Signal zur Phasen-Synchronisation an den CLKFB-Eingang des DCMs zurückgekoppelt.

Das 50 MHz Taktsignal wird mittels Frequenzsynthese am CLKFX-Port erzeugt. Die Frequenz von 50 MHz wird durch die Generics „CLKF_DIVIDE“ und „CLKF_MULTIPLY“ wie folgt eingestellt:

$$CLKX\,(50MHz) = (\tfrac{2}{4} \times CLKIN\,(100MHz))$$

Durch die Synthese des Designs wurde die Instanz des DCMs zu der, in der Abbildung 52 dargestellten, Schaltung verdrahtet.

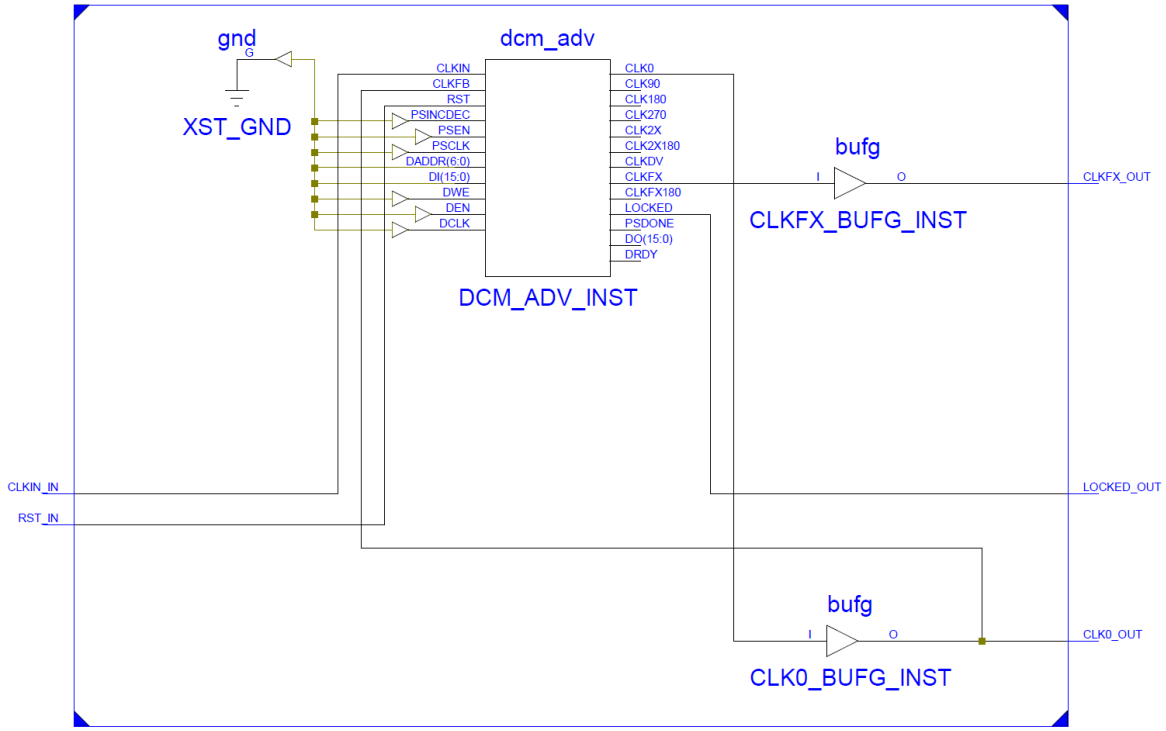


Abbildung 52: Instanz des Digital Clock Managers

5.1.3.3 Takt Multiplexing

Zum glitchfreien Umschalten zwischen zwei Taktsignalen verfügt der Virtex5 FPGA über spezielle Taktmultiplexer. Diese sind die „BUFGCTRL“-Grundkomponenten auch Primitive genannt. [4] S.66 Der BUFGCTRL ist ein globaler Taktbuffer welcher über zwei Takt- und mehrere Steuereingänge verfügt um einen der beiden Eingangstaktsignale glitchfrei zum Ausgang des Taktmultiplexers durchzuschalten. Das BUFGCTRL-Element ist in der Abbildung 53 symbolisch dargestellt.

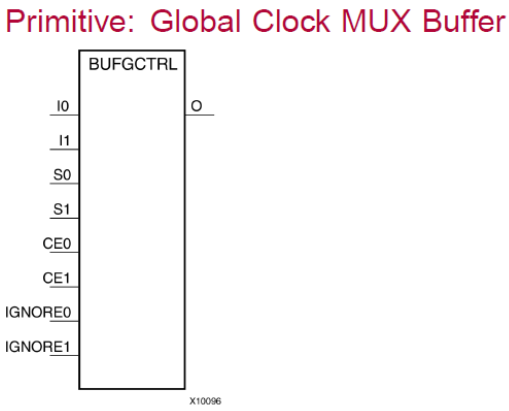


Abbildung 53: Globaler Taktmultiplexer. Quelle [4] S.66

Das BUFGCTRL- Element wird wie folgt in der ISE-Umgebung instanziiert:

```
1     BufGCtrlMux_l : BUFGCTRL
2     generic map (
3           INIT_OUT     => 0,
4           PRESELECT_I0 => FALSE,   -- TRUE/FALSE set the I0 input after configuration
5           PRESELECT_I1 => FALSE)   -- TRUE/FALSE set the I1 input after configuration
6     port map (
7           O       => Muxed_Clk,      -- Clock MUX output
8           CE0     => not Clk_Sel_0,  -- Clock enable0 input
9           CE1     => Clk_Sel_0,      -- Clock enable1 input
10          I0      => 100MHz_Clk,     -- Clock0 input
11          I1      => 50MHz_Clk,      -- Clock1 input
12          IGNORE0 => '0',            -- Ignore clock select0 input
13          IGNORE1 => '0',            -- Ignore clock select1 input
14          S0      => '1',            -- Clock select0 input
15          S1      => '1'             -- Clock select1 input
16    );
```

Abbildung 54: Instanz des BUFGCTRL- Elementes in der ISE-Umgebung

In der, in Abbildung 54 gezeigten, Instanziierung arbeitet das BUFGCTRL-Element synchron. Das Umschalten der Taktsignale erfolgt durch das Aktivieren und Deaktivieren der beiden Takteingänge „CE0“ und „CE1“ (Clock Enable) des Multiplexers. Die synchrone Implementierung des BUFGCTRL-Elements ist geeignete um glitchfrei zwischen permanent vorhandenen Eingangstaktsignalen zu umzuschalten. Sollten aber die Taktsignale zeitweise unterbrochen sein, so besteht die Gefahr eines sogenannten Deadlocks. Da der Multiplexer sicher stellen muss, dass die beiden Takteingangsignale im Schaltzeitpunkt Pegel „0“ aufweisen. Sollte einer der beiden Taktsignale nicht anliegen, so wird das Umschalten nicht möglich. Aufgrund dessen eignet sich die oben aufgelistete synchrone Implementierung des Multiplexers nicht zum Umschalten des SPI-Taktsignals. Deshalb wird

für diesen Takt ein weiteres BUFGCTRL-Element im asynchronen Modus durch folgende Instantiierung Implementiert:

```
1      BufGCtrlMux_2 : BUFGCTRL
2      generic map (
3             INIT_OUT     => 0,
4             PRESELECT_I0 => FALSE,   -- TRUE/FALSE set the I0 input after configuration
5             PRESELECT_I1 => FALSE)   -- TRUE/FALSE set the I1 input after configuration
6      port map (
7             O       => MDX_In_Clk,     -- Clock MUX output
8             CE0     =>'1',             -- Clock enable0 input
9             CE1     =>'1',             -- Clock enable1 input
10            I0      => Muxed_Clk,      -- Clock0 input
11            I1      => SPI_Clk,        -- Clock1 input
12            IGNORE0 => '1',            -- Ignore clock select0 input
13            IGNORE1 => '1',            -- Ignore clock select1 input
14            S0      => not Clk_Sel_1,  -- Clock select0 input
15            S1      => Clk_Sel_1       -- Clock select1 input
16     );
```

Abbildung 55: Instanz des BUFGCTRL- Elementes in der ISE-Umgebung

Hier erfolgt das Umschalten der Eingangstaktsignale mittels Select-Eingänge „S0“ und „S1“. Die beide Clock-Enable Eingänge „CE0“ und „CE1“ sind hier statisch auf logische „1“ geschaltet und somit ständig aktiv. Durch das Setzen der „IGNORE0“ und „IGNORE1“ auf logische „1“ sind die Algorithmen zum synchronen Umschalten der Taktsignale deaktiviert.

Software Entwicklung und Integration

Die Software des PowerPC 440 Prozessors wurde ausgehend aus Messergebnissen des Praxisprojektes II [7] in der Single-Threaded Umgebung realisiert. Dies ermöglicht die Implementierung des LwIP-Stacks im RAW API Modus, wodurch die maximal mögliche Datenübertragungsbandbreite des Gigabit Ethernet Interfaces erreicht werden kann.

In der Software des PowerPC440 Prozessors werden folgende Aufgaben realisiert:

- Initialisierung und Betrieb aller in der Abbildung 22 dargestellten IP-Cores die durch den PLB mit dem eingebetteten Prozessor Blocks verbunden sind.
- Der Betrieb von zwei von einander unabhängigen LwIP-Basierte TCP/IP-Listenern, wobei ein Listener zum Empfangen der Befehle vom PC vorgesehen ist und der zweite zum Übertragen der Matrixdaten mittels Gigabit Interface verwendet wird.
- Interpretieren und Ausführen der vom PC Empfangenen Befehle
- Realisierung von mehreren Readout-Modi des Lambda-Moduls

Die vorhandenen Softwaremodule sind in der Abbildungen 56 und 57 dargestellt. Wobei die Abbildung 56 den Projekt Explorer zeigt und die Abbildung 57 stellt die Zusammenhänge der Softwaremodule dar.

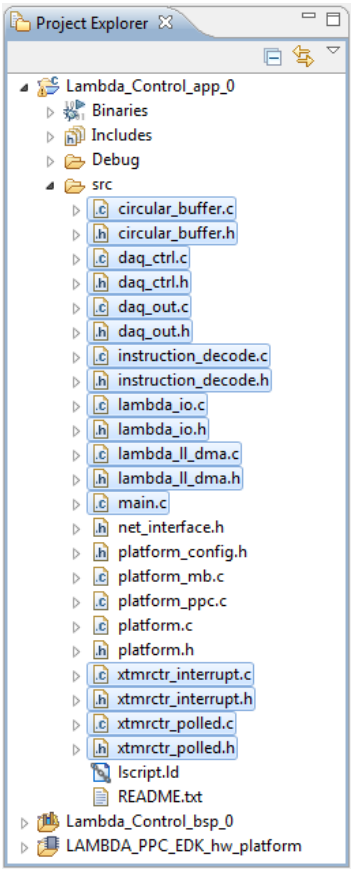


Abbildung 56: Datei Explorer des Software Projektes

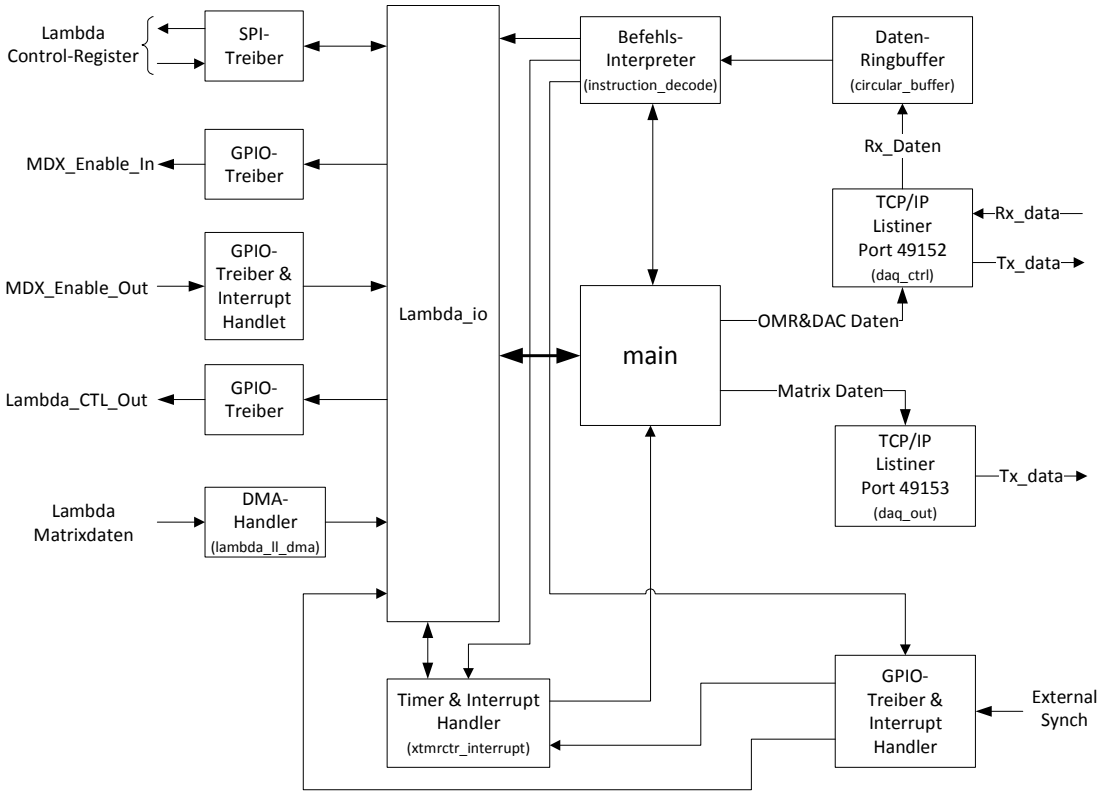


Abbildung 57: PowerPC 440 Software Konzept

Die zentrale Einheit des Programms ist die Datei *„main.c“* und vor allem seine *„main“* Funktion, in der die Initialisierungen aller IP-Cores und Software Module erfolgt. Des Weiteren erfolgen dort die Aufrufe der Funktionen des *„Lambda-IO“* Muduls zum Lesen und Schreiben der Register- und Matrixdaten der Medipix3 Chips und Übergabe der gelesenen Daten an die *„Daq_Ctrl“* und *„Daq_Out“* Server zur weiteren Übertragung mittels TCP/IP zum PC. Das *„Lambda-IO“* Softwaremodul verwendet wiederum die durch „Application Programming Interface“ der Single-Threaded Standalone Umgebung zur Verfügung stehenden Treiber für die Kommunikation mit der Peripherie-Komponenten des eingebetteten Systems.

Als konzeptionelle Umsetzung des Prozessor Programms wurde die Trennung in Interrupt Service Routinen (ISR) und Polling Loop eingesetzt. Wobei die Flags in der Mainfunktion zyklisch abgefragt und die entsprechenden Operationen je nach gesetztem Flag ausgeführt werden. Die Ereignisse lösen die Interrupts aus, wobei in der ISR die notwendigen, zeitkritischen Aktionen behandelt werden und die Flags, für die weitere, zeitlich unkritische Abarbeitung der Aktionen in der Mainfunktion gesetzt werden. Hierfür wurde in der „instruction_decode.h“ Datei folgende „*LAMBDA_CONTROL*“ Datenstruktur angelegt:

```
1  typedef struct LAMBDA_CONTROL{
2
3  volatile char OMR[16];              //Char Array zum Speichern der OMR-Daten
4  volatile int MDX_Number;            //Nummer des aktuellen Medipix3 Chips
5  volatile int image_count;           //Anzahl der Images die gelesen werden
                                         sollen
6  volatile Xboolean instruction_decode;//Flag, initiiert Decodierung der
                                          Befehle
7  volatile Xuint32 ShutterTime;       //Definiert die Zeit einer Bildaufnahme
8  volatile int ImageSelect;           //Dient der Flusssteuerung der
                                         Bildaufnahme
9  volatile Xboolean  matrix_load;     //Flag, initiiert das Laden der Chip-
                                         Matrix
10 volatile int daq_mode;              //Definiert das Readout-Modus
11 volatile Xboolean op_started;       //Signalisiert gestartete Operation
12 volatile Xboolean rx_rdout_started; //Signalisiert gestartete Readout-
                                         Operation
13 volatile Xboolean rx_rdout_done;    //Signalisiert abgschlossene Readout-
                                         Operation
14 volatile int cnt_nr;                //Nummer des Medipix3 Counters
15 volatile Xboolean tmr_start_readout;//Flag, Initiiert Readout eines Images
16 volatile Xboolean ext_synch;        //Flag, Aktiviert Externe Synchronisation
17 volatile int interface;             //Definiert Ethernet Interface (1G oder
                                         10G)
18 }lambda_control_t;
```

Abbildung 58: „Lambda_Control“ Datenstruktur der „instruction_decode.h“ Datei

Die „*LAMBDA_CONTROL*“ Datenstruktur fasst verschiedene Datentypen zusammen, die für die Steuerung des Detektors verwendet werden.

Die Funktionsweise der *„main“* Funktion wird durch das Funktionsdiagramm der Abbildung 59 gezeigt.

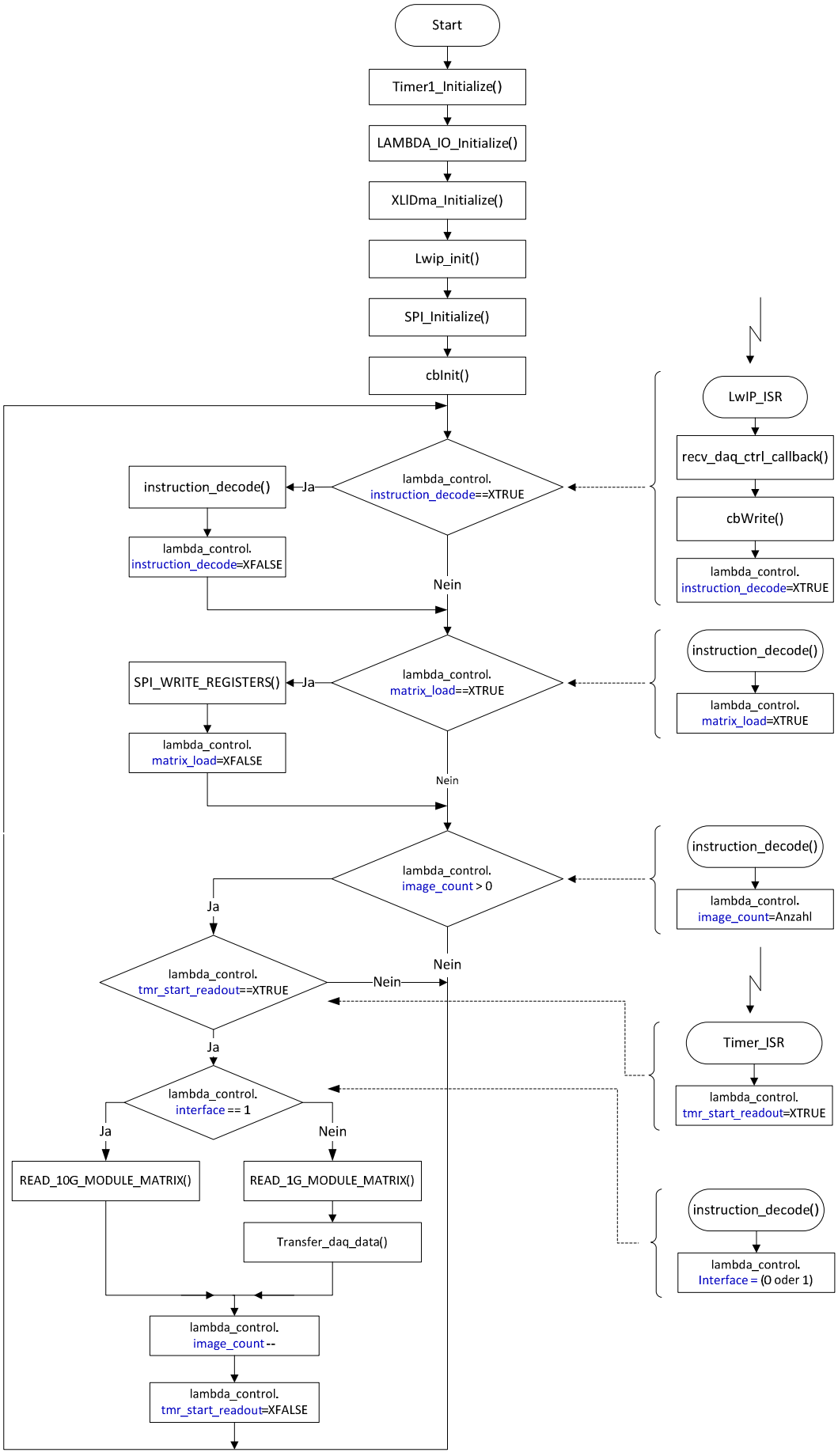


Abbildung 59: Funktionsdiagramm zur Darstellung der Funktionsweise der Mainfunktion

Zu Beginn der Mainfunktion erfolgen die Initialisierungen folgender Einheiten:

- Initialisierung des Timers durch die „*Timer1_Initialize*“ Funktion
- Initialisierung aller vier GPIO’s durch den Aufruf der „*LAMBDA_IO_Initialize*“ Funktion
- Initialisierung des LocalLink Interfaces durch die „*XLlDma_Initialize*“ Funktion
- Initialisierung des SPI durch die „*SPI_Initialize*“ Funktion
- Initialisierung des Ringbuffers durch die „*cbInit*“ Funktion

In der Endlosschleife der Mainfunktion werden folgende Flags der „*LAMBDA_CONTROL*“ Datenstruktur zyklisch abgefragt:

- *instruction_decode*: Dieser Flag wird in der *„recv_daq_ctrl_callback“* Funktion gesetzt. Dies geschieht wenn eine oder mehrere Befehle vom PC empfangen und die Daten in den Ringbuffer geschrieben wurden. Durch das Setzen dieses Flags erfolg ein Aufruf der *„instruction_decode“* Funktion, wodurch das Dekodieren der empfangenen Befehle vorgenommen wird. Nach dem Ausführen der *„instruction_decode“* Funktion wird das *„instruction_decode“* Flag wieder gelöscht.
- *matrix_load*: Das *„matrix_load“* Flag wird in der Funktion *„instruction_decode“* gesetzt, wenn bei der Dekodierung der empfangenen Befehle der Befehl zum Laden der Medipix3 Chip Matrix erkannt wird. Durch das Setzen dieses Flags wird die *„SPI_WRITE_REGISTERS“* Funktion aufgerufen und die Chip Matrix wird mittels SPI mit den Matrixdaten aus dem Ringbuffer initialisiert.
- *image_count*: Durch die *„image_count“* Variable wird die Auslese und die nachfolgende Übertragung der ausgelesenen Imagedaten zum PC gesteuert. Die *„image_count“* Variable wird in der *„instruction_decode“* Funktion entsprechend der gewünschten Anzahl von Bilden initialisiert. Wobei die Anzahl der Bilder die Ausgelesen werden sollen ebenfalls per Befehl von PC aus festgelegt wird. Ist der Inhalt der Variable größer als null, beginnt das Readout der Bilder. Die Geschwindigkeit des Readouts wird durch den Timer1 gesteuert, hierfür erzeugt der Timer Interrupts beim Überlauf, wobei in deren ISR der *„tmr_start_readout“* Flag gesetzt wird. Daraufhin erfolgt die Unterscheidung der Variabel *„interface“*, welche das Ethernet Interface zum Übertragen der Daten definiert. Entsprechend dieser Variablen wird die Funktion *„READ_10G_MODULE_MATRIX“* zum Auslesen und Übertragen der Matrixdaten mittels 10G Ethernet- oder *„READ_1G_MODULE_MATRIX“* Funktion zum Auslesen und nachfolgende Übertragung der Matrixdaten mittels 1G Interface aufgerufen. Zuletzt wird die Variable *„image_count“* um eins dekrementiert und das *„tmr_start_readout“* Flag gelöscht.

5.1.4 Implementierung von LwIP - Basierten Server Applikationen

Zur Realisierung der Datenkommunikation über Gigabit Ethernet sind zwei LwIP-basierte TCP/IP-Listener *„daq_ctrl“* und *„daq_out“* unter Verwendung des RAW-API[38] in der Software des Prozessors implementiert. Der RAW-API Modus erlaubt die Verwendung des LwIP-Stacks in einer Standalone Umgebung. Die Funktionsweise und Implementierung der LwIP-Basierten Applikationen wurde im Praxisprojekt II Quelle [7] weitergehend diskutiert.

Nach dem Server-Client Model werden die beiden oben genannten TCP/IP-Listener im weiteren Text als Server bezeichnet.

Die Abbildung 60 Zeigt das Funktionsprinzip der Datenkommunikation zwischen Detektor und PC mittels Gigabit Interface.

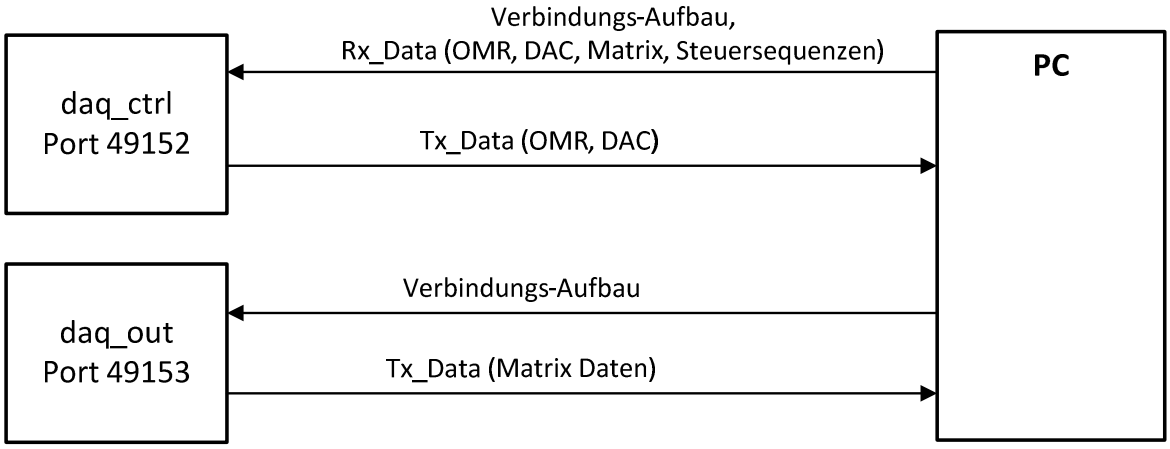


Abbildung 60: Datenkommunikation über Gigabit Interface

Der *„daq_ctrl“* Server öffnet passiv den Port 49152 und wartet auf eingehenden Verbindungen sowie der Übertragung von Steuersequenzen, von OMR-, DAC- und Matrixdaten. Beim Empfangen eines IP-Paketes wird die Funktion *„recv_daq_ctrl_callback“* der *„daq_ctrl.c“* Datei Aufgerufen. Dort werden die empfangenen Daten mittels der Funktion *„cbWrite“* in den Ringbuffer geschrieben. Als Argumente bekommt die Funktion *„cbWrite“* einen Pointer auf den Ringbuffer, einen Pointer auf den Payload des empfangenen TCP-Segmentes und als drittes Argument die Menge der empfangenen Daten. Nach dem Schreiben der Daten in den Ringbuffer wird die Flag *„instruction_decode“* der *„lambda_control“* Instanz gesetzt.

```
1 err_t recv_daq_ctrl_callback(void *arg, struct tcp_pcb *tpcb,
2                               struct pbuf *p, err_t err)
3 {
4     /* do not read the packet if wey are not in ESTABLISHED state */
5     if (!p) {
6           tcp_close(tpcb);
7           tcp_recv(tpcb, NULL);
```

[38] Application Programming Interface

```
8             return ERR_OK;
9       }
10      /* indicate that the packet has been received */
11      tcp_recved(tpcb, p->len);
12
13      cbWrite(&ring_buffer, p->payload, p->len);
14
15      if(lambda_control.matrix_load == 0 ){
16            lambda_control.instruction_decode=1;
17            //Stop taking the images then receiving a new command.
18            lambda_control.image_count = 0;
19      }
20      /* free the received pbuf*/
21      pbuf_free(p);
22
23      return ERR_OK;
24 }
```

Abbildung 61: Recv_daq_ctrl_callback - Funktion des Daq_ctrl Servers

Durch den *„daq_ctrl“* Server werden auch die ausgelesenen DAC-Daten der Medipix3 Chips zum PC übertragen. Zum Lesen und Übertragen der DAC-Daten werden die in der Abbildung 62 dargestellten Funktionen direkt aus der *„instruction_decode“* Funktion aufgerufen:

```
1 //Read Medipix3 Registers.
2 int Bytes = SPI_READ_REGISTERS(SpiInstancePtr,(Xuint8 *)response_buffer,
3                                                  lambda_control.MDX_Number);
4
5 response_daq_data((char *)response_buffer,Bytes);
```

Abbildung 62: Funktionsaufrufe zum Lesen und Übertragen der Medipix3 Register

Die Übertragung der durch die *„SPI_READ_REGISTERS“* Funktion ausgelesenen Register Daten erfolgt durch die, in der *„daq_ctrl.c“* Datei implementierte, *„response_daq_data“* Funktion. Als Argumente werden der Funktion einen Pointer auf Datenbuffer und deren Länge übergeben.

Der *„daq_out“* Server ist nach gleichem Prinzip wie der *„daq_ctrl“* Server implementiert und wird zum Übertragen der Matrixdaten zum PC verwendet. Der *„daq_out“* Server nutzt den Port 49153. Nach dem Verbindungsaufbau, der vom Client initiiert wird, können die Matrixdaten mittels der *„transfer_daq_data“* Funktion übertragen werden. Der Quellcode der *„transfer_daq_data“* Funktion stellt die Abbildung 63 dar.

```
1 int transfer_daq_data(struct netif *netif, char* SendbufferPrt, int length) {
2
3      int copy = 0;
4      err_t err;
5      struct tcp_pcb *tpcb = connected_daq_out_pcb;
6      //Number of Bytes to be send ones by tcp_write.
7      int TRANSMIT_BUFSIZE = 1460;
8      int sent_data = 0;
9      int rest = 0;
10
11     if (!connected_daq_out_pcb)
12           xil_printf("No Connection:  %d\n\r", connected_daq_out_pcb);
13           return ERR_OK;
14
15     do{
16           if(tcp_sndbuf(tpcb) > TRANSMIT_BUFSIZE) {
17                 err = tcp_write(tpcb,SendbufferPrt,TRANSMIT_BUFSIZE, copy);
18                 if (err != ERR_OK) {
```

```
19                          xil_printf("daq: Error on tcp_write: %d\n\r", err);
20                                  connected_daq_out_pcb = NULL;
21                                  return -1;
22                                  break;
23                  }
24                  tcp_output(tpcb);
25
26                  SendbufferPrt += TRANSMIT_BUFSIZE;
27                  sent_data += TRANSMIT_BUFSIZE;
28                  rest = length - sent_data;
29                  if(rest < TRANSMIT_BUFSIZE) TRANSMIT_BUFSIZE = rest;
30
31          }
32          xemacif_input(netif);
33
34    }while(sent_data < length);
35
36   return 0;
37 }
```

Abbildung 63: Transfer_daq_data - Funktion zum Übertragen der Matrixdaten

Zum Übertragen der Matrixdaten benötigt die *„transfer_daq_data“* Funktion drei Argumente: einen Pointer auf die globale *„netif“*[39] Struktur, einen Pointer auf den Datenbuffer der die Matrixdaten enthält und die Länge des Datenbuffers selbst. Zu Beginn wird die Variable *„TRANSMIT_BUFSIZE“* mit dem Wert 1460 initialisiert, was der Größe des MSS[40] [7] S.11 eines TCP-Sigmentes entspricht. Durch den Aufruf der Funktion *„tcp_sndbuf“* wird überprüft ob noch genug freier Speicherplatz im Send-Buffer der TCP-Instanz vorhanden ist. Ist der Rückgabewert der *„tcp_sndbuf“* Funktion größer als der eines MSS, wird mit der Übertragung eines TCP-Segmentes begonnen. Die Übertragung der Daten erfolgt durch die Aufrufe der Funktionen *„tcp_write“* und *„tcp_output“*. Weiter erfolgen die Inkrementierung des *„SendbufferPrt“* um die Anzahl der gesendeten Daten und die Ermittlung des Restes der noch zur Übertragung stehenden Matrixdaten. Wird der Rest kleiner als der eines MSS, wird die Variable *„TRANSMIT_BUFSIZE“* mit dem Wert der *„rest“* Variabler initialisiert. Durch diese Initialisierung wird die Übertragung der Daten, die sich unmittelbar hinter den Matrixdaten im Speicher befinden, verhindert. Die vorher beschriebene Übertragungsprozedur wird zyklisch wiederholt solange die Menge der gesendeten Daten kleiner als die Länge des Datenbuffers ist, der die zu übertragenden Daten enthält. Die Länge des Datenbuffers wird durch die Variable *„length“* der *„transfer_daq_data“* Funktion übergeben. Die Funktion *„xemacif_input“* gibt die empfangenen Acknowledgement-Sequenzen der gegenüber liegenden TCP-Instanz dem LwIP-Stack weiter.

5.1.5 Befehlsinterpreter

Der Befehlsinterpreter wird in der Funktion *„instruction_decode“* realisiert, deren Quellcode in der Datei *„instruction_decode.c“* gefunden werden kann. Der Befehlsinterpreter unterscheidet grundsätzlich drei Arten der Befehlssequenzen:

---

[39] Die Instanz der Struktur „netif“ enthält alle Parameter der Netzwerkschnittstelle, wie z.B. die MAC-Adresse, die MTU, den Typ der Verbindung und deren Geschwindigkeit.

[40] Maximum Segment Size. Quelle [7, S.10]

- Steuersequenzen zum Initialisieren der *„lambda_control“* Struktur und Ausführen der Reset-Operationen der Medipix3 Chips
- OMR-Datensequenz
- Execution- sowie DAC- und Matrix-Datensequenz

Der Ringbuffer wird 16-Bitweise gelesen und interpretiert. Im Beispiel der Abbildung 64 wird ein Codeausschnitt der *„instruction_decode“* Funktion zum Lesen und Interpretieren der Befehlssequenzen gezeigt.

```
1 void instruction_decode(XSpi *SpiInstancePtr, circularBuffer_t *cbuffer){
2
3      Xint32 subIndex = 0;
4      char subbuffer[16];
5      char elem;

6      /* Remove all elements from a ring buffer*/
7      while(!cbIsEmpty(&ring_buffer))
8      {
9         cbRead(cbuffer, &elem);
10        subbuffer[subIndex] = elem;
11        subIndex++;
12
13        if(subIndex == 16)
14        {
15            subIndex = 0;
16
17            if((subbuffer[0] == 0x00000000) && (subbuffer[1] == 0x000000F0))
18            {
19               // Auswerten der Steuersequenzen und Initialisierung der
20                  Lambda_control Struktur
                  ..
21            }
22            else if((subbuffer[0] == 0x00) && (subbuffer[1] == 0x30))
23            {
24               // Speichern der OMR-Daten in dem OMR-Array der
25                  lambdada_control Struktur;
                  ..
26            }
27            else if((subbuffer[0] == 0xa0) || (subbuffer[1] == 0xa0))
28            {
29               // Auswerten und ausführen der, durch OMR definierten
30                  Operationen.
                  ..
31            }
32         }
33     }
34
35 }
```

Abbildung 64: Codeausschnitt der Instruction_decode - Funktion zum Lesen und Interpretieren der Befehlssequenzen

Das Interpretieren der Befehlssequenzen erfolgt Anhand der Header-Sequenz welche durch die ersten zwei Bytes der Sequenz definiert ist. Die Daten des Ringbuffers werden zyklisch gelesen und interpretiert, solange die, in der Zeile 7 der Abbildung 64 gezeigte, Funktion „cbIsEmpty“ einen „false“ Wert zurückliefert. Bei der Rückgabe von „true“ wird

dieser Vorgang beendet, da in diesem Fall keine Daten mehr im Ringbuffer vorhanden sind.

Die Steuersequenzen werden durch die Bytefolge (0x00, 0xF0) als Header eingeleitet, das Format der Steuersequenzen stellt die Tabelle 9 dar.

Byte 0	Byte 1	Byte 2	Byte 3	Byte 4	Byte 5
Header		Befehlsdefinition	Befehlsparameter		
0x00	0xF0	0x01 - Medipix Reset	-	-	-
0x00	0xF0	0x02 - Fast Matrix Clear	-	-	-
0x00	0xF0	0x32 - Aktivieren alle Medipixe	-	-	-
0x00	0xF0	0x33 - Aktivieren Medipix 1	-	-	-
0x00	0xF0	0x34 - Aktivieren Medipix 2	-	-	-
0x00	0xF0	0x35 - Aktivieren Medipix 3	-	-	-
0x00	0xF0	0x36 - Aktivieren Medipix 4	-	-	-
0x00	0xF0	0x37 - Aktivieren Medipix 5	-	-	-
0x00	0xF0	0x38 - Aktivieren Medipix 6	-	-	-
0x00	0xF0	0x33 - Aktivieren Medipix 7	-	-	-
0x00	0xF0	0x34 - Aktivieren Medipix 8	-	-	-
0x00	0xF0	0x35 - Aktivieren Medipix 9	-	-	-
0x00	0xF0	0x36 - Aktivieren Medipix 10	-	-	-
0x00	0xF0	0x37 - Aktivieren Medipix 11	-	-	-
0x00	0xF0	0x38 - Aktivieren Medipix 12	-	-	-
0x00	0xF0	0x46 - Setzen Timerperiode (in Timer Ticks, 10ns/Tick)	MSB		LSB
0x00	0xF0	0x47 - Image Count	MSB		LSB
0x00	0xF0	0x48 - Externe Synchronisation	„0x01“- aktiviert „0x00“- deaktiviert	-	-
0x00	0xF0	0x49 - Readout Mode	„0x01“ - sequential „0x02“ - continues	-	-
0x00	0xF0	0x50 – Interface	„0x00“- 1 Gigabit/s „0x01“- 10 Gigabit/s	-	-

Tabelle 9: Format der Steuersequenzen

Alle Befehlssequenzen müssen mindestens 16-Byte lang sein, damit der Befehlsinterpreter diese fehlerfrei interpretieren kann. Zum Beispiel um den Reset der Medipix3 Chips auszuführen wird entsprechend der Tabelle 9 folgende Befehlssequenz gesendet:

"0x00 0xF0 0x01 0x00 0x00 0x00"

nachfolgend werden alle Steuersequenzen mit 10 leeren Bytes aufgefüllt um die gewünschte Länge von 16 Bytes zu erreichen. Der nachfolgende Codeausschnitt zeigt die Auswertung der Steuersequenzen am Beispiel der ersten und letzten Sequenzen der Tabelle 9.

```
1     if((subbuffer[0] == 0x00000000) && (subbuffer[1] == 0x000000F0))
2     {
3           switch (subbuffer[2])
4           {
5                 case 0x01:// Medipix Reset command.
6                       MDX_Reset(SpiInstancePtr);
7                 break;
8
9                 ..
```

```
10
11                  case 0x50:// Interface select 1G/10G.
12                        lambda_control.interface = subbuffer[3];
13                  break;
14            }
15
16      }
```

Abbildung 65: Codeausschnitt der Instruction_decode - Funktion zum Auswertung der Steuersequenzen

Die OMR-Datensequenz wird durch die Bytefolge (0x00, 0x30) eingeleitet. Wird diese Bytefolge als Header erkannt, erfolgt einen Kopiervorgang entsprechend der Abbildung 66, welcher die, aus dem Ringbuffer gelesenen 16 Datenbytes in den OMR-Buffer der *„lambda_control“* Struktur kopiert.

```
1       else if((subbuffer[0] == 0x00) && (subbuffer[1] == 0x30))
2       {
3             Xint32 Index;
4             for ( Index = 0; Index < 16; Index++ )// We store the OMR Data.
5             {
6                   lambda_control.OMR[Index] = subbuffer[Index];
7             }
8       }
```

Abbildung 66: Codeausschnitt der Instruction_decode - Funktion zum Kopieren der OMR-Daten

Zum Ausführen der durch die OMR-Daten definierten Operation wird eine Execution-Sequenz verwendet. Diese wird durch den 0xA0 Byte eingeleitet, wobei das 0xA0 Zeichen als erstes oder zweites Element des *„subbuffer“* Arrays stehen darf. Die Execution-Sequenz kann auch die DAC- oder Matrixdaten enthalten die in den Medipix3 Chip geladen werden müssen. Wird die Execution-Sequenz erkannt, erfolgt die Auswertung der so genannten „M“-Bits der, in dem OMR-Buffer gespeicherten, OMR-Daten. Die Funktionsweise der „M“-Bits des OMR-Registers ist in der Tabelle 1 des ersten Kapitels Beschrieben. Das Auswerten der „M“-Bits und Ausführen der wichtigsten Chip-Operation wird durch den Codeausschnitt der Abbildung 67 gezeigt.

```
1       else if((subbuffer[0] == 0xa0) || (subbuffer[1] == 0xa0))
2       {
3          Xuint32 Bytes;
4          // We mask the M-Bits of OMR Register.
5          INSTRUCTION = lambda_control.OMR[3] & 0xE0;
6
7          switch (INSTRUCTION)
8          {
9              ..
10
11             case 0xC0://Read DAC command.
12                   Bytes = SPI_READ_REGISTERS(SpiInstancePtr,(Xuint8*)
13                               response_buffer, lambda_control.MDX_Number);
14                   response_daq_data((char *)response_buffer,Bytes);
15             break;
16
17             case 0x80://Set DAC command.
18                   SPI_WRITE_REGISTERS(SpiInstancePtr, cbuffer, 36,
19                                                lambda_control.MDX_Number);
20             break;
21
22             ..
23
24             case 0x00:// Read Matrix command counter0.
```

```
                    ..
25                  // Starten das Lesevorgang der Matrix-Daten
                    ..
26            break;

27            case 0x20:// Read Matrix command counter1.
                    ..
28                  // Starten das Lesevorgang der Matrix-Daten
                    ..
29            break;

30            case 0x40:// Load Counter 0 command
31                  lambda_control.matrix_load = 1;
32            break;
33
34            case 0x60:// Load Counter 0 command.
35                  lambda_control.matrix_load = 1;
36            break;
37        }
38    }
```

Abbildung 67: Codeausschnitt der Instruction_decode - Funktion zum Auswerten der „M“-Bits und Ausführen der Chip-Operation

Die „M“-Bits sind im vierten Element des OMR-Buffers lokalisiert und sind die drei MSB[41] dieses Elementes. Zum Auswerten der Information dieser Bits wird das vierte Element des OMR-Buffers, entsprechend der Zeile 5 in Abbildung 67, mit 0xE0 maskiert und das Resultat dieser logischen Verknüpfung wird in der Variablen „*INSTRUCTION*“ gespeichert. Weiter folgt die Fallunterscheidung des Inhaltes dieser Variablen mit der „Switch-Case“ Anweisung.

Werden die Zeichen 0xC0 oder 0x80 als „M“-Bits erkannt, erfolgen die Lade- bzw. Lesevorgänge der DAC-Register des Medipix3 Chips. Zum Laden und Lesen der Chip Register werden die in der *„lambda_io.c“* Datei implementierten *„SPI_READ_REGISTERS“* und *„SPI_WRITE_REGISTERS“* Funktionen eingesetzt. Diese beiden Funktionen verwenden die SPI-Schnittstelle für die serielle Datenkommunikation mit den Medipix3 Chips. Zum Laden der DAC-Register der Chips wird die *„SPI_WRITE_REGISTERS“* Funktion aufgerufen, wobei durch diese Funktion das, in der Abbildung 7 des ersten Kapitels dargestellten Protokoll zum Laden der DAC-Daten realisiert wird. Hingegen wird zum Lesen der DAC-Register die *„SPI_READ_REGISTERS“* Funktion verwendet. Diese Funktion realisiert das in der Abbildung 8 dargestellten Protokoll.

Werden die Zeichen 0x00 oder 0x20 als „M“-Bits erkannt, beginnt das Lesen der Pixel-Matrix der Medipix3 Chips. Dieser Vorgang wird durch den, in der Abbildung 68 gezeigten Codeausschnitt gestartet.

```
1 switch (INSTRUCTION)
2 {
3   ..
4
5   case 0x00://Read Matrix command.
```

[41] Most Signifikant Bit

```
6
7      lambda_control.image_count = imageCount;
8
9      if(lambda_control.daq_mode == RXCONTRW_AQ){
10
11        SETUP_INT_BASED_RX_READOUT(SpiInstancePtr);
12
13        if(lambda_control.ext_synch == 1){
14            /* Enable LAMBDA_CTRL_IN interrupt  if external synchronization is
15            defined */
16            XGpio_InterruptEnable(&LAMBDA_CTRL_IN, LAMBDA_CTRL_IN_INTERRUPT);
17        }
18        else{
19            /* Disable LAMBDA_CTRL_IN interrupt  if external synchronization is
10            not defined */
21            XGpio_InterruptDisable(&LAMBDA_CTRL_IN,LAMBDA_CTRL_IN_INTERRUPT);
22
23            /* Start the timer if external synchronization is not enabled */
24            TmrCtrStart(&xps_timer_1_Timer, TIMER_CNTR_1,
25                                                  lambda_control.ShutterTime);
26        }
27     }else if(lambda_control.daq_mode == SEQUEN_AQ){
28        SETUP_SEQUN_RX_READOUT(SpiInstancePtr);
29     }
30    break;
31
32    ..
33 }
```

Abbildung 68: Codeausschnitt der Instruction_decode - Funktion zum Lesen der Pixel-Matrix

Zum Starten des Lesevorganges wird die Variable *„image_count“* der *„lambda_control“* Struktur initialisiert. Die Funktionsweise dieser Variablen ist bereits am Anfang dieses Kapitels auf Seite 69 beschrieben. Weiter folgt die Unterscheidung des festgelegten Readout Modus. Es sind zweit Readout Modi implementiert: Continuous und Sequential. Der Unterschied dieser beiden Readout Modi wurde im Kapitel 2.2.4.6 gezeigt. Wird der Continuous Readout Modus eingestellt, erfolgt die Vorbereitung des Readout Prozesses mit Hilfe der *„SETUP_INT_BASED_RX_READOUT“* Funktion, wobei durch diese Funktion das OMR der Medipix3 Chips geladen wird. Des Weiteren wird geprüft ob die externe Synchronisation verwendet werden soll. Wird die externe Synchronisation nicht verwendet, wird die Interrupt Annahme für „*LAMBDA_CTRL_IN*“ GPIO durch den Aufruf der *„XGpio_InterruptDisable“* Funktion gesperrt und anschließend der Timer 1 gestartet. Der Timer 1 wird für die Steuerung der Readout Geschwindigkeit verwendet. Soll die externe Synchronisation verwendet werden, so wird die Interrupt Annahme für „*LAMBDA_CTRL_IN*“ GPIO zugelassen. Der Timer 1 wird in diesem Fall in der Interrupt Service Routine des *LAMBDA_CTRL_IN*“ GPIOs gestartet, wenn die Pegeländerung am externen Synchronisationseingang erkannt wird.

Schließlich wird durch 0x40 und 0x60 als „M“-Bits das Laden der Matrix gestartet im dem die als Flag verwendete *„matrix_load“* Variable der *„lambda_control“* Struktur gesetzt wird.

5.1.6 Das Lesen der Pixel-Matrix

Das Lesen der Pixel-Matrix im Continuous Readout Mode erfolgt wie bereits beschrieben, in Abhängigkeit von dem eingestellten Ethernet Interface durch die Aufrufe der

„READ_1G_RX_MODULE_MATRIX“ oder *„READ_10G_RX_MO-DULE_ MATRIX“* Funktionen aus der *„main“* Funktion. Die beiden oben genannten Funktionen sind Bestandteile der *„lambda_io.c“* Datei. Die Abbildung 69 zeigt das Funktionsablaufdiagramm der *„READ_1G_RX_MODULE_MATRIX“* Funktion. Zu Beginn dieser Funktion wird das LocalLink Interface der Parallel-Readout Komponente aktiviert und das Readout des Lambda Moduls durch die, in Abbildung 69 gezeigte „*START_INT_BASED_RX_READOUT“* Funktion gestartet. Als nächstes wird der Timer 0 initialisiert und gestartet. Weiter folgt die zyklische Abfrage des *„rx_rdout_done“* Flags der *„lambda_control“* Struktur und der *„XTmrCtr_IsExpired“* Funktion. Wird die *„rx_rdout_done“* Flag gesetzt, wird die Warteschleife beendet und die „*FINALIZE_INT_BASED_RX_READOUT“* Funktion aufgerufen. Dadurch wird geprüft, ob die Matrixdaten mittels LocalLink Interface fehlerfrei in das DDR2-SDRAM des eingebetteten Systems geschrieben wurden. Das *„rx_rdout_done“* Flag wird in der Interrupt Service Routine des *„MDX_ENB_OUT*“ GPIOs gesetzt. Diese Interrupt Service Routine wird ausgeführt, wenn die Medipix3 Chips, den abgeschlossenen Readout-Prozess durch „Low“ Pegel der „Enable_Out“ Ausgänge quittieren.

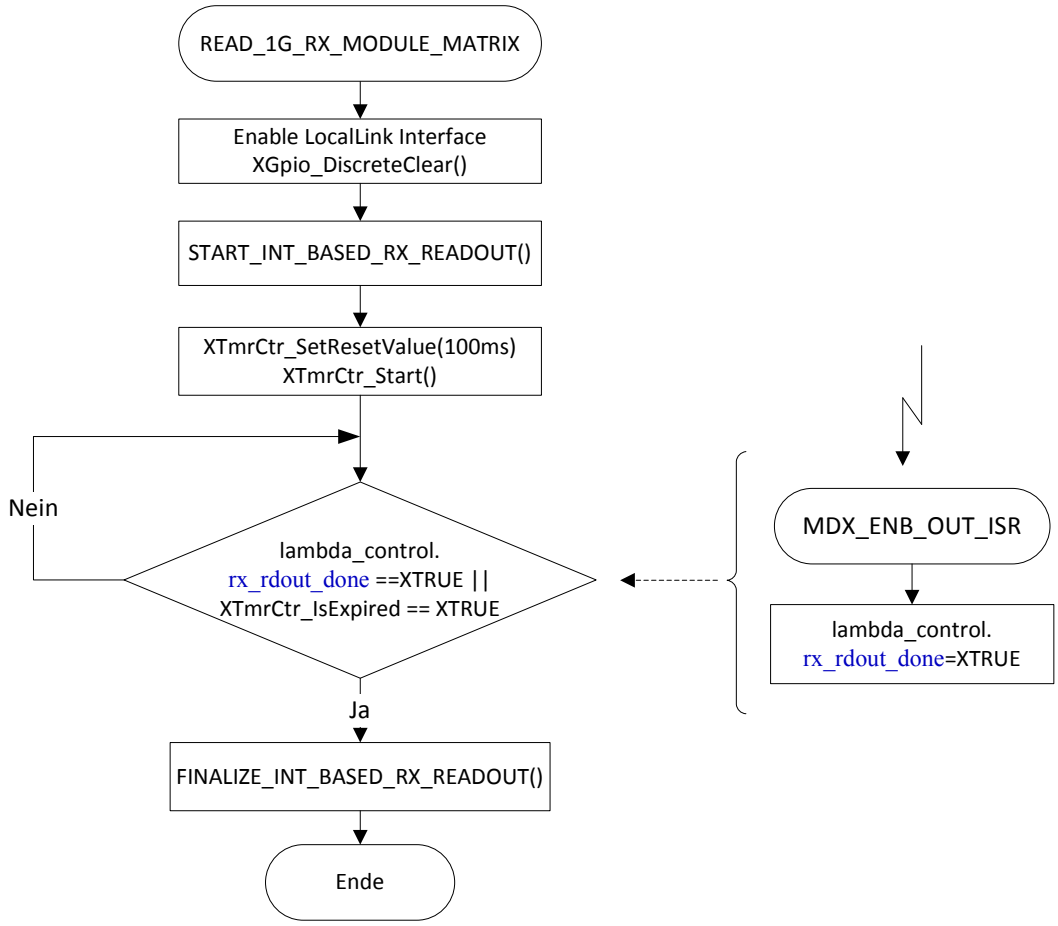


Abbildung 69: Funktionsablaufdiagramm der READ_1G_RX_MODULE_MATRIX-Funktion

Sollte der, mit einem 100ms Intervall initialisierte, Timer 0 ablaufen, ohne dass die Medipix3 Chips den Readout quittieren, bedeutet dies ein Fehler im System. In diesem Fall

wird die Warteschleife der *„READ_1G_RX_MODULE_MATRIX“* Funktion ebenfalls verlassen und eine Fehlermeldung durch das USART[42] des Systems ausgegeben. Der Timer verhindert damit, dass das Systems hängen bleibt wenn die Medipix3 Chips die ausgeführte Operation nicht mehr quittieren sollten.

Zum Starten des Readouts wird die *„rx_rdout_done“* Flag zu Beginn der *„START_INT_BASED_RX_READOUT“* Funktion gelöscht und die „Shutter1_CounterSel“ Eingänge der Medipix3 Chips getoggelt. Dies erfolgt durch das Setzen bwz. Löschen des 10-ten Bits des *„GPIO_DATA“* Registers des *„LAMBDA_CTRL_OUT“* GPIOs, mit Hilfe der *„XGpio_DiscreteClear“* und *„XGpio_DiscreteSet“* Funktionen. Diese Funktionen ermöglichen die Bitoperationen mit GPIO-Register durch das Read-Modify-Write Verfahren.

```
1  void START_INT_BASED_RX_READOUT(XSpi *SpiInstancePtr)
2  {
3      Xuint8 exe[] = {0xa0,0x00};
4      lambda_control.rx_rdout_done = XFALSE;
5
6      /*Here we toggle a counter select input*/
7      if(lambda_control.cnt_nr == 0){
8             XGpio_DiscreteClear(&LAMBDA_CTRL_OUT, CHANNEL, 1 << COUNTER_SEL );
9             lambda_control.cnt_nr = 1;
10     }
11     else{
12            XGpio_DiscreteSet(&LAMBDA_CTRL_OUT, CHANNEL, 1 << COUNTER_SEL );
13            lambda_control.cnt_nr = 0;
14     }
15
16     /*Enable all Medipixes*/
17     XGpio_DiscreteWrite(&MDX_ENB_IN, CHANNEL, 0x000);
18
19     /*Enable SPI-Clock*/
20     XGpio_DiscreteSet(&LAMBDA_CTRL_OUT, CHANNEL, 1 << CLK_SELECT );
21
22     /*Send the EXE-Sequence over SPI-Interface*/
23     XSpi_Transfer(SpiInstancePtr, (Xuint8 *)exe, 0, 1);
24
25     /*Disable SPI-Clock*/
26     XGpio_DiscreteClear(&LAMBDA_CTRL_OUT, CHANNEL, 1 << CLK_SELECT );
27
28     /*Set the image size using slave register 1*/
29     LAMBDA_LL_DMA_mWriteSlaveReg1(XPAR_LAMBDA_LL_DMA_0_BASEADDR, 0,
30                                                                IMAGE_SIZE);
31
32     /*Enable LLink Logic, LSB of register 0 enables the Local-Link FSM*/
33     LAMBDA_LL_DMA_mWriteSlaveReg0(XPAR_LAMBDA_LL_DMA_0_BASEADDR, 0,
34                                                                0x80000000);
35
36     lambda_control.rx_rdout_started = XTRUE;
37
38 }
```

Abbildung 70: Quellcode der START_INT_BASED_RX_READOUT - Funktion

Des Weiteren werden alle 12 Medipix3 Chips durch das Schreiben des „0x000“ Wertes in das *„GPIO_DATA“* Registers des *„MDX_ENB_IN“* GPIOs mit dem Aufruf der *„XGpio_DiscreteWrite“* Funktion, in der Zeile 17 der Abbildung 70, aktiviert. Durch die Aufrufe der, in der Zeilen 20, 23 und 26 gezeigten Funktionen, erfolgt die Umschaltung des

[42] Universal Synchronous and Asynchronous Serial Receiver and Transmitter

Takt Multiplexers und Übertragung der „0xA0“ Execution-Sequenz mittels SPI. Durch die Übertragung dieser Sequenz wird der eigentliche Lesevorgang der Pixel-Matrix gestartet. Durch die *„LAMBDA_LL_DMA_mWriteSlaveReg1“* Funktion erfolgt ein Schreibzugriff auf dem Slave-Register 1 des „Lambda_ll_Interfaces“ wodurch die Größe eines Lambda Images von 1180032 Byte durch das dritte Argument *„IMAGE_SIZE“* dieser Funktion festgelegt wird. Zum Aktivieren des Zustandsautomaten des „Lambda_ll_Interfases“ wird das Bit „0“ des Slave-Registers0 durch die *„LAMBDA_LL_DMA_mWriteSlaveReg0“* Funktion gesetzt. Die Funktionsweise des „Lambda_ll_Interfaces“ ist im Kapitel 5.1.2 beschrieben.

Zum Abschließen des Lesevorganges wartet die in der Zeile 6 der Abbildung 71 gezeigte *„CheckDmaResult“* Funktion bis die gestartete DMA-Transaction beendet ist.

```
1  Xuint32 FINALIZE_INT_BASED_RX_READOUT()
2  {
3      Xuint32  Bytes = 0;
4      int Status = 0;
5
6      Status = CheckDmaResult(&LlDma);
7             if (Status != XST_SUCCESS) {
8                    xil_printf("Failed rx_data receive\n\r");
9             }
10
11     /* Read the Payload Counter which is connected to Slave Register 3*/
12     Bytes = LAMBDA_LL_DMA_mReadSlaveReg3(XPAR_LAMBDA_LL_DMA_0_BASEADDR, 0);
13
14     /* Disable LLink Logic, LSB of register 0 disables the Local-Link FST*/
15     LAMBDA_LL_DMA_mWriteSlaveReg0(XPAR_LAMBDA_LL_DMA_0_BASEADDR, 0,
16                                                                  0x00000000);
17     /*Disable all Medipixes*/
18     XGpio_DiscreteWrite(&MDX_ENB_IN, CHANNEL, 0xffffffff);
19
20     BuferAdrsses[0] = (u32)RxPacketAddr;
21
22     lambda_control.rx_rdout_done = XTRUE;
23
24     return Bytes;
25
26 }
```

Abbildung 71: Quellcode der FINALIZE_INT_BASED_RX_READOUT- Funktion

Durch den Lesezugriff auf das Slave-Register 3 mittels der *„LAMBDA_LL_DMA_mReadSlaveReg3“* Funktion wird der Zählerstand des Payload Zählers ermittelt, welcher der Menge der empfangenen Daten entspricht. Anschließend wird durch das Löschen des ersten Bits „0“ im Slave-Register 0 der Zustandsautomat des „Lambda_ll_Interfaces“ in den IDLE-Zustand versetzt. Schließlich werden alle 12 Medipix3 Chips deaktiviert und das *„rx_rdout_done“* Flag gesetzt wird.

Der Aufbau der *„READ_10G_RX_MODULE_MATRIX“* Funktion ist etwas einfacher als der, der *„READ_1G_RX_MODULE_MATRIX“* Funktion. Unter Verwendung des 10 Gigabit Ethernet Interfaces erfolgt die Übertragung der Matrixdaten parallel zum eingebetteten System. Deshalb sind alle Funktionsaufrufe die mit dem „Lambda_ll_Interface“ IP-Core zusammenhängen nicht in der *„READ_10G_RX_MODULE_MATRIX“* Funktion enthalten. Der Quellcode der *„READ_10G_RX_MODULE_MATRIX“* Funktion kann in der *„lambda_io.c“* Datei gefunden werden.

5.1.7 Externe Synchronisation des Detektors

Durch die externe Synchronisation soll der Readout Prozess der Bilder durch eine steigende Flanke des Signals an dem Synchronisationseingag gestartet werden. Diese Möglichkeit ist mit Hilfe der *„LAMBDA_CTRL_IN“* GPIO realisiert. Hierfür ist das Bit „0“ des *„GPIO_DATA“* Registers durch den GPIO-Port mit einem Pin eines Steckverbinders verbunden. Durch eine Pegeländerung am diesen Pin wird ein Interrupt erzeugt und die dem *„LAMBDA_CTRL_IN“* GPIO zugeordnete *„LAMBDA_CTRL_IN_ISR“* Routine eingesprungen. Die Abbildung 72 zeigt ein Codeausschnitt dieser Routine.

```
1  void LAMBDA_CTRL_IN_ISR(void *InstancePtr)
2  {
3   XGpio *GpioPtr = (XGpio *)InstancePtr;
4
5   ..
6
7   /* Here Process the interrupt. */
8    int input = XGpio_DiscreteRead(&LAMBDA_CTRL_IN, CHANNEL);
9    if(input == 0x00000001){
10     /* Open the shutter here if external synchronization is enabled */
11     XGpio_DiscreteClear(&LAMBDA_CTRL_OUT, CHANNEL, 1 << SCHUTTER );
12
13     /* Start the timer if external interrupt on LAMBDA_CTRL_IN occurs */
14     TmrCtrStart(&xps_timer_1_Timer, TIMER_CNTR_1, lambda_control.ShutterTime);
15
16     /* Clear the interrupt such that it is no longer pending in the GPIO */
17     (void)XGpio_InterruptClear(GpioPtr, LAMBDA_CTRL_IN_INTERRUPT);
18
19   }
20   ..
21 }
```

Abbildung 72: Codeausschnitt der Interrupt Service Routine des „LAMBDA_CTRL_IN“ GPIOs

Im Gegensatz zu anderen Systemen ermöglicht der Xilinx XPS keine präzise Definition der Interrupt Bedingungen. Es ist nicht möglich die Interrupt Generierung durch einen hohen oder niedrigen Pegel bzw. eine steigende oder fallende Flanke des Signals an einem bestimmten Pin des GPIOs zu definieren. Stattdessen wird der Interrupt durch jede Änderung an jedem Pin des GPIOs erzeugt. Aus diesem Grund wird in der Interrupt Service Routine der Abbildung 72 erst durch den Lesezugriff auf das *„GPIO_DATA“* Register mittels der *„XGpio_DiscreteRead“* Funktion, sichergestellt, dass der Interrupt durch die steigende Signalflanke am externen Synchronisationseingang generiert wurde. Dies führt allerdings zu einer Erhöhung der Reaktionszeit des Systems bei der externen Synchronisation.

Zum starten des Lesevorganges der Bilder werden die *„Shutter“* Eingänge der Medipix3 Chips, entsprechend der Zeile 11 der Abbildung 72, aktiviert und der Timer 1 gestartet. Schließlich wird durch den Aufruf der *„XGpio_InterruptClear“* Funktion das Bit „31“ des „Interrupt Status Registers“ gelöscht um eine wiederholte Interrupt Annahme durch das *„LAMBDA_CTRL_IN“* GPIO zu vermeiden.

6 Testergebnisse

In diesem Kapitel werden die Testergebnisse des entwickelten Systems zusammengefasst. Es wird die gemessene Reaktionszeit des Interrupt Systems dargestellt und die Vorgehensweise zum Ermitteln dieser Zeit beschrieben. Des Weiteren folgt die Beschreibung der Messmethode und der Messergebnisse der Readout Geschwindigkeit.

Reaktionszeit des Interrupt Systems

Der Test der Reaktionszeit des Interrupt Systems ist zum Ermitteln der Reaktionszeit des Systems auf eine Pegel Änderung am externen Synchronisationseingang interessant. Zum Durchführen dieses Tests wurde das *„Shutter“* Signal, welches in der Interrupt Service Routine des *„LAMBDA_CTRL_IN“* GPIOs (Abbildung 72) aktiviert wird, durch die FPGA Firmware nach draußen herausgeführt. Zusammen mit dem Eingangssignal des externen Synchronisationseingangs wird das *„Shutter“* Signal mit Hilfe eines Oszilloskops gemessen. Die Ergebnisse dieser Messung stellt die Abbildung 73 dar.

Abbildung 73: Oszillogramm mit Darstellung der gemessenen Reaktionszeit des Interrupt Systems

Durch den unteren, cyanfarbenen Graphen in der Abbildung 73 wird das am externen Synchronisationseingang anliegende Signal dargestellt. Der gelbe, obere Graph stellt den Verlauf des *„Shutter“* Signals dar. Die Zeitdifferenz zwischen der steigenden Flanke des am externen Eingang anliegendes Signals und der fallenden Flanke des *„Shutter“* Signals beträgt 401µs und entspricht der Reaktionszeit des Interrupt-Systems.

Messung der Image Readout Geschwindigkeit

Für die genaue Messung der Image Readout Geschwindigkeit wird die Zeit die zum auslesen und Übertragen eines Images benötigt wird gemessen. Hierfür wird die zweite Timer-Einheit des Timer 1 IP-Cores verwendet, wobei diese Timer Einheit in einem Polling-Modus zur Testzwecken implementiert wird, aber im regulären Betrieb des Detektors nicht verwendet wird. Zum Messen der oben genannten Zeiten wird die *„main“* Funktion des Prozessorprogramms wie folgt ergänzt:

```
1  /*-----------Time Measurement------------*/
2  startTimeMeasurement();
3  /*---------------------------------------*/
4  if(lambda_control.interface == 1){
5
6      READ_10G_RX_MODUL_MATRIX(&SpiInstance);
7  }
8  else{
9
10     READ_1G_RX_MODUL_MATRIX(&SpiInstance);
11     transfer_daq_data(netif, (char *)BuferAdrsses[0],1180032);
12 }
13 /*-----------Time Measurement------------*/
14 getElapsedTime();
15 /*---------------------------------------*/
```

Abbildung 74: Codeausschnitt der main- Funktion zum Messen der Readout Zeiten

Durch die, in der Zeile 2 gezeigte *„startTimeMeasurement“* Funktion wird die Messung der Zeit gestartet. Hierfür beinhaltet diese Funktion folgende Funktionsaufrufe:

```
1      /*---------Time measurement ----------------------------------------*/
2      /* Get a snapshot of the timer counter value before it's started.*/
3      Value1 = XTmrCtr_GetValue(&TimerCounter, 0);
4
5      /* Start the timer counter such that it's incrementing by default*/
6      XTmrCtr_Start(&TimerCounter, 0);
7      /*-------------------------------------------------------------------*/
```

Abbildung 75: Codeausschnitt der startTimeMeasurement - Funktion

Die Variable *„Value1“* wird mit dem Anfangswert des Timers initialisiert und anschließend wird der Timer gestartet. Nach dem Auslesen und der Übertragung der Matrixdaten, Zeilen 6, 10 und 11 der Abbildung 74, wird der Timer durch die *„getElapsedTime“* Funktion, wie folgt angehalten und ausgewertet:

```
1      /* Get a snapshot of the timer counter value before it will be stopped */
2      Value2 = XTmrCtr_GetValue(&TimerCounter, 0);
3
4      /* Stop the Counter */
5      XTmrCtr_Stop(&TimerCounter, 0);
6
```

```
7       /* Reset the Counter */
8       XTmrCtr_Reset(&TimerCounter, 0);
```

Abbildung 76: Codeausschnitt der getElapsedTime – Funktion, zum Auslesen des Timer Counters

wobei die Variable *„Value2“* den Zählstand des Timer Zählers annimmt. Des Weiteren wird der Timer für den nächsten Messvorgang durch einen Reset vorbereitet. Zum Berechnen der verstrichenen Zeit wird entsprechend der Abbildung 77 ein Differenz der Variablen *„Value1“* und *„Value2“* gebildet und in Millisekunden dargestellt.

```
1       /* Berechnen der Übertragungszeit in ms. */
2       time = (Value2-Value1)*0.00001;
3       sprintf(string, "%2.4f", time);
4       xil_printf("\r\n Elapsed time: %s ms. \r", string);
```

Abbildung 77: Codeausschnitt der getElapsedTime – Funktion, zum Berechnen und Ausgeben der gemessenen Zeit

Zum Ausgeben der Messergebnisse durch das USART wird die, in der Zeile 4 der Abbildung 77 gezeigte, *„xil_printf“* Funktion verwendet.

Zum Ermitteln der Readout Geschwindigkeit wurde zuerst das Ethernet-Interface auf 1G eingestellt, so dass die Matrixdaten mittels TCP/IP über Gigabit-Interface zum PC übertragen werden. Anschließend wurden 20 Bilder aufgenommen und die dafür benötigte Zeit ausgegeben. Die Abbildung 78 zeigt die Ergebnisse dieser Zeitmessung.

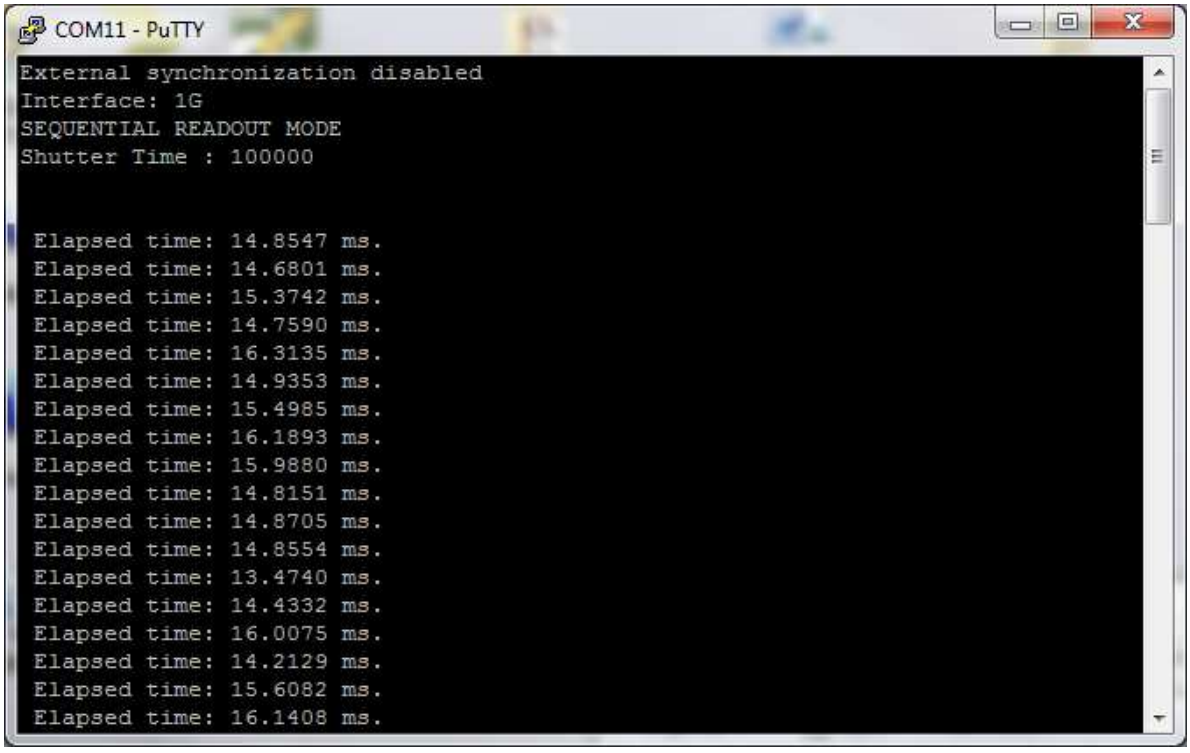


Abbildung 78: Ergebnisse der Zeitmessung mit 1G Ethernet-Interface

Der Mittelwert der benötigten Zeit, zum Auslesen und Übertragen eines Images, beträgt in dieser Messung etwa 14,94 ms. Die daraus resultierende Readout Geschwindigkeit ergibt sich zu:

$$\boxed{(14{,}94ms)^{-1} = 66{,}95 Bilder / s}$$

Gleichung 15

das ist um 5,65 Bilder/s weniger als mit der Gleichung 7 des Kapitels 4.14.1 annähernd berechnet wurde.

Zum Ermitteln der Readout Geschwindigkeit mit Verwendung des 10G Ethernet-Interfaces zum Übertragen der Matrixdaten wurde das Ethernet-Interface dementsprechend auf 10G eingestellt und die oben beschriebene Zeitmessung wiederholt. Die Abbildung 79 stellt die Ergebnisse dieser Zeitmessung dar.

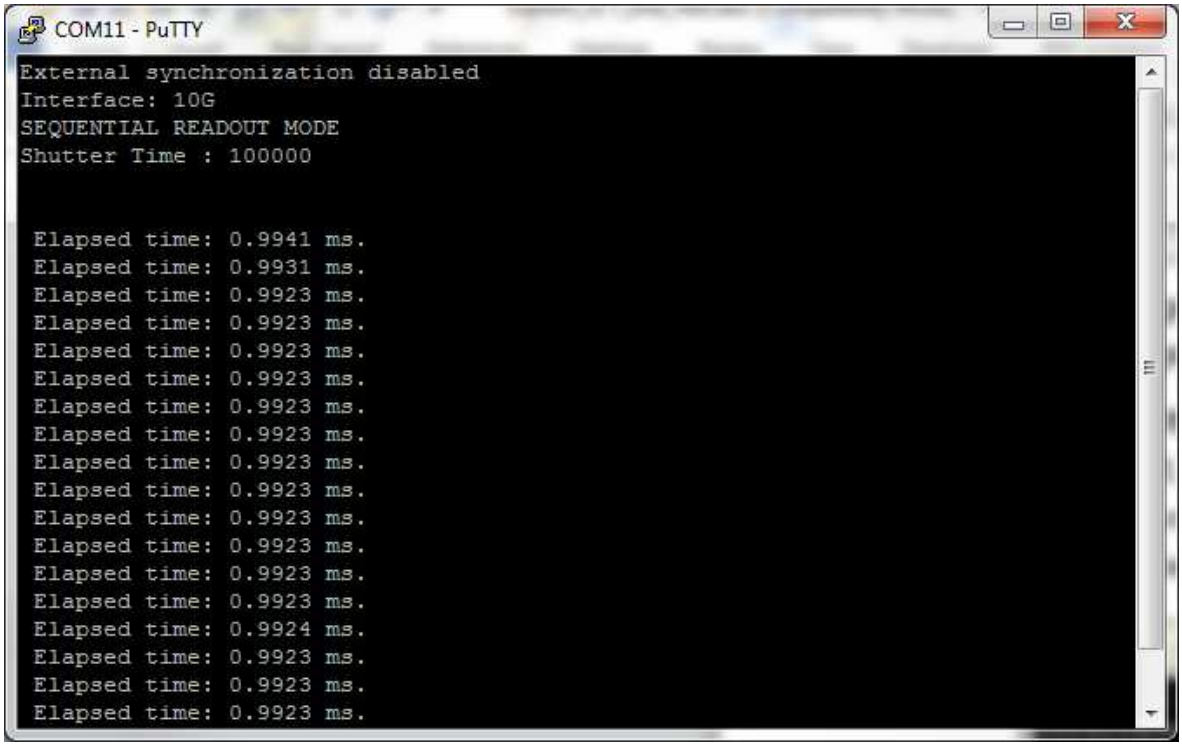


Abbildung 79: Ergebnisse der Zeitmessung mit 10G Ethernet-Interface

Der Mittelwert der benötigten Zeit, zum Auslesen eines Images, beträgt in diesem Fall etwa 1 ms. Dies entspricht einer Readout Geschwindigkeit von 1000 Bildern/s Die Übertragung der Daten zum PC erfolgt in diesem Fall parallel zum eingebetteten System.

7 Zusammenfassung und Ausblick

Die Entwicklung der Firmware-Einheiten des Lambda Projektes wurde erfolgreich abgeschlossen. Entsprechend der Messergebnissen der Readout Geschwindigkeit wurde die maximale Bildwiederholungsfrequenz von 1000 Bildern/s erreicht. Des Weiteren wurden die Firmware-Komponenten sowie Software-Module zum Realisieren der externen Synchronisation implementiert und getestet.

Das Detektor-System wurde mehrmals in Experimenten eingesetzt, dabei sind keine gravierenden Probleme aufgetreten. Zukünftig sind einige Erweiterungen der Firmware durch Implementierung von weiteren Readout Modi und Synchronisationsmöglichkeiten des Detektors geplant.

Zum Verbessern der Detektor-Eigenschaften können folgende Schritte unternommen werden: Die Performance der seriellen Datenkommunikation zwischen dem eingebetteten System und den Medipix3 Chips die, für die Konfiguration und Ansteuerung der Chips verwendet wird, kann durch Einsatz von schnelleren Komponenten anstatt des etwas langsameren SPI's, verbessert werden. Zum Beispiel könnte der nicht verwendete Tx-Kanal des LocalLink Interfaces mit nachfolgender Datenserialisierung zur Übertragung der Daten zu den Medipix3 Chips verwendet werden. Dabei wird die serielle Datenübertragung unter Verwendung derselben Taktquelle möglich. Dadurch entfällt die Notwendigkeit des in Kapitel 5.1.3 beschriebenen, Umschaltens zwischen den SPI- und DCM Takten.

Des Weiteren könnte die Reaktionszeit des Systems auf externe Trigger verbessert werden. Hierfür könnte der, in dieser Implementierung nicht verwendete, „EICC440CRITIRQ"-Eingang des PowerPC 400 Prozessors für kritische Interrupts verwendet werden. Dabei würde der „LAMBDA_CTRL_IN" GPIO ohne Verwendung eines Interrupt Kontrollers direkt an diesen Eingang angeschlossen. Damit wird unter anderem auch die Anzahl der Zugriffe des Prozessors auf Register der Peripherie Komponenten zum Ermitteln der Interruptquelle vermindert, weil an den „EICC440CRITIRQ"-Eingang nur eine Interruptquelle angeschlossen wird.

Literatur

[1] R. Ballabriga, X. Llopart: Medipix3 Manual v1.4
CERN, 10/31/2012

[2] http://www.xilinx.com/tools/platform.htm, verfügbar am 15.09.2013, 18:20

[3] Embedded Processor Block in Virtex-5 FPGAs, Reference Guide UG200 (v1.8) February 24, 2010
http://www.xilinx.com/support/documentation/user_guides/ug200.pdf, verfügbar am 15.09.2013, 18:20

[4] Virtex-5 Libraries Guide for HDL Designs
http://www.xilinx.com/itp/xilinx10/books/docs/virtex5_hdl/virtex5_hdl.pdf, verfügbar am 15.09.2013, 18:20

[5] Virtex-5 FPGA User Guide UG190 (v5.4) March 16, 2012
http://www.xilinx.com/support/documentation/user_guides/ug190.pdf, verfügbar am 15.09.2013, 18:20

[6] Ulrike Gebert: Diplomarbeit,
„Untersuchung von Eigenschaften Photonenzählender pixelierter Halbleiterdetektoren der Medipix-Familie“
http://www.pi4.nat.uni-erlangen.de/novel-detectors/publications/phd-diplomathesis/gebert_diplom.pdf, verfügbar 14.10.2013, 18:43

[7] Smoljanin Sergej: Praxisprojekt II,
„Implementierung und Performance Test eines LwIP-Netzwerkstaks auf dem Integrierten PowerPC Prozessor eines Xilinx Virtex5 FPGAs“

[8] Embedded System Tools Reference Manual, UG111 (v13.4) January 18, 2012,
http://www.xilinx.com/support/documentation/sw_manuals/xilinx13_4/est_rm.pdf, verfügbar 14.10.2013, 18:43

[9] PPC440x5 CPU Core User’s Manual,
https://www-01.ibm.com/chips/techlib/techlib.nsf/techdocs/622DCFA1D98B1F10002575A7005464AF/$file/ppc440x5_um.pdf, verfügbar 14.10.2013, 18:43

[10] 128-Bit Processor Local Bus Architecture Specifications Version 4.7,
https://www-01.ibm.com/chips/techlib/techlib.nsf/techdocs/3BBB27E5BCC165BA87256A2B0064FFB4/$file/PlbBus_as_01_pub.pdf, verfügbar 14.10.2013, 18:43

[11] IBM PowerPC 440 Embedded Core, Product Brief,

https://www-01.ibm.com/chips/techlib/techlib.nsf/techdocs/F72367F770327F8A87256E63006CB7EC/$file/PowerPC440_Nov2006.pdf, verfügbar am 14.10.2013, 18:33

[12] MicroBlaze Processor Reference Guide, Embedded Development Kit, EDK 13.4, UG081 http://www.xilinx.com/support/documentation/sw_manuals/xilinx13_4/mb_ref_guide.pdf, verfügbar am 15.09.2013, 18:20

[13] MicroBlaze Processor Performance, http://www.xilinx.com/tools/microblaze.htm, verfügbar am 15.09.2013, 18:20

[14] XPS General Purpose Input/Output (GPIO) (v2.00.a) LogiCORE IP http://www.xilinx.com/support/documentation/ip_documentation/xps_gpio.pdf, verfügbar am 15.09.2013, 18:20

[15] XPS Serial Peripheral Interface (SPI) (v2.02a) LogiCORE IP, http://www.xilinx.com/support/documentation/ip_documentation/xps_spi.pdf, verfügbar am 15.09.2013, 18:20

[16] http://photonscience.desy.de/research/technical_groups/detectors/projects/Lambda/index_eng.html

[17] Tri-Mode Ethernet MAC Block Diagram Quelle: LogiCore IP XPS LL TEMAC (v2.03a), http://www.xilinx.com/support/documentation/ip_documentation/xps_ll_temac.pdf, verfügbar am 15.09.2013, 18:20

[18] System ACE CompactFlash Solution, DS080 (v2.0) October 1, 2008 http://www.xilinx.com/support/documentation/data_sheets/ds080.pdf, verfügbar am 15.09.2013, 18:20

[20] Platform Specification Format Reference Manual, UG642 March 1, 2011 http://www.xilinx.com/support/documentation/sw_manuals/xilinx13_4/psf_rm.pdf, verfügbar am 15.09.2013, 18:20

[21] XPS Interrupt Controller (v2.01a), LogiCORE IP, http://www.xilinx.com/support/documentation/ip_documentation/xps_intc.pdf, verfügbar am 15.09.2013, 18:20

[22] EDK Concepts, Tools and Techniques, (UG683) EDK 11 http://www.xilinx.com/support/documentation/sw_manuals/xilinx11/edk_ctt.pdf, verfügbar am 15.09.2013, 18:20

[23] Virtex-5 Family Overview, DS100 (v5.0) February 6, 2009, http://www.xilinx.com/support/documentation/data_sheets/ds100.pdf, verfügbar am 14.10.2013, 18:09

Anlagen

Anlagen, Teil 1

Bei der nachfolgenden Auflistung handelt es sich um den Quellcode des im Kapitel 5.1.2 beschriebenen „Lambda_ll_interface“ IP-Cores.

```
-- DO NOT EDIT BELOW THIS LINE --------------------
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;

library proc_common_v3_00_a;
use proc_common_v3_00_a.proc_common_pkg.all;

-- DO NOT EDIT ABOVE THIS LINE --------------------

--USER libraries added here
------------------------------------------------------------------------------
-- Entity section
------------------------------------------------------------------------------
entity user_logic is

  generic
  (
    -- DO NOT EDIT BELOW THIS LINE ---------------------
    -- Bus protocol parameters, do not add to or delete
    C_SLV_DWIDTH                   : integer := 32;
    C_NUM_REG                      : integer := 4
    -- DO NOT EDIT ABOVE THIS LINE ---------------------
  );

  port
  (
    -- ADD USER PORTS BELOW THIS LINE ------------------
    Ll_Clk                              : in  std_logic;
    Ll_Rst                              : in  std_logic;
    tx_data                             : in  std_logic_vector(0 to 31);
    tx_rem                              : in  std_logic_vector(0 to 3);
    tx_sof_n                            : in  std_logic;
    tx_eof_n                            : in  std_logic;
    tx_sop_n                            : in  std_logic;
    tx_eop_n                            : in  std_logic;
    tx_src_rdy_n                        : in  std_logic;
    tx_dst_rdy_n                        : out std_logic;
    rx_data                             : out std_logic_vector(0 to 31);
    rx_rem                              : out std_logic_vector(0 to 3);
    rx_sof_n                            : out std_logic;
    rx_eof_n                            : out std_logic;
    rx_sop_n                            : out std_logic;
    rx_eop_n                            : out std_logic;
    rx_src_rdy_n                        : out std_logic;
    rx_dst_rdy_n                        : in  std_logic;
    chip_scope_data                     : out std_logic_vector(31 downto 0);
    chip_scope_flags                    : out std_logic_vector(15 downto 0);
    fifo2ll_din                         : in  std_logic_vector(31 downto 0);
    fifo2ll_fifo_empty                  : in  std_logic;
    ll2fifo_rd_enable                   : out std_logic;
    ll2fifo_clk                         : out std_logic;
    ll2rdout_enable                     : out std_logic;
    rdout2ll_ready                      : in  std_logic;
```

```
    --USER ports added here
    -- ADD USER PORTS ABOVE THIS LINE ------------------
  );
end entity user_logic;

------------------------------------------------------------------------------
-- Architecture section
------------------------------------------------------------------------------

architecture IMP of user_logic is

--USER signal declarations added here, as needed for user logic

--LocalLink signals
signal sig_rx_sof_n             :std_logic:= '0';
signal sig_rx_eof_n             :std_logic:= '0';
signal sig_rx_sop_n             :std_logic:= '0';
signal sig_rx_eop_n             :std_logic:= '0';
signal sig_rx_src_rdy_n         :std_logic:= '0';
signal sig_rx_dst_rdy_n         :std_logic:= '0';
signal rx_pkt_cnt               :std_logic:= '0';
signal sig_rx_src_sw            :std_logic:= '0';

-- Payload counter signals
Signal   payload_cnt_rst        :std_logic:= '0';
signal    payload_cnt_enb       :std_logic:= '0';
signal    payload_wrt_count     :std_logic_vector(0 to 31):=(others =>'0');

-- Payload state signals
signal wr_payload_data    :std_logic:='0';

-- External fifo signals
signal sig_ll2fifo_enable : std_logic:='0';

-- Footer counter signals
signal footer_count       :std_logic_vector(3 downto 0):=(others =>'0');
signal footer_cnt_enb     :std_logic;
signal foot_cnt_rst       :std_logic;

-- Footer state signals
signal footer_data        :std_logic_vector(0 to 31);
signal wr_footer_data     :std_logic:='0';

--FSM enable signal
signal Ll_Enable :std_logic:='0';

-- States definition
type STATES is (IDLE, RX_SOF, RX_SOP, WRITE_DATA, RX_EOP, FOOTER, RX_EOF,
WAIT_FOR_END); --Aufzählungstyp

signal STATE, NEXT_STATE: STATES;--Prozess-Kommunikation

-----------------------------------------
-- Signals for user logic slave model s/w accessible register example
-----------------------------------------
signal slv_reg0                       : std_logic_vector(0 to C_SLV_DWIDTH-1);
signal slv_reg1                       : std_logic_vector(0 to C_SLV_DWIDTH-1);
signal slv_reg2                       : std_logic_vector(0 to C_SLV_DWIDTH-1);
signal slv_reg3                       : std_logic_vector(0 to C_SLV_DWIDTH-1);

begin

--USER logic implementation added here

--Rx signals
rx_sof_n           <=     sig_rx_sof_n;
rx_eof_n           <=     sig_rx_eof_n;
rx_sop_n           <=     sig_rx_sop_n;
rx_eop_n           <=     sig_rx_eop_n;
rx_src_rdy_n       <=     sig_rx_src_rdy_n;
sig_rx_dst_rdy_n   <=     rx_dst_rdy_n;
```

```
-- DMA enable by lsb of slave rgister0
Ll_Enable           <= slv_reg0(0);


-- Readout enable
ll2rdout_enable     <= Ll_Enable;
ll2fifo_clk         <= ll_clk;

-----------------------------------------
-- DMA Implimentation                   --
-----------------------------------------
SYNC_PROC: process(Ll_Clk, Ll_Rst) --Zustandsaktualisierung
begin
   if Ll_Rst = '1' then STATE <= IDLE;
         elsif Ll_Clk = '1' and Ll_Clk'event then
                        STATE <= NEXT_STATE;
   end if;
end process SYNC_PROC;

NEXT_STATE_DECODE: process (sig_rx_sof_n, sig_rx_eof_n, sig_rx_sop_n,
sig_rx_eop_n, sig_rx_dst_rdy_n,sig_rx_src_rdy_n, payload_cnt_enb,
payload_wrt_count,footer_count, Ll_Enable, STATE)
begin

      -- Signals set to one <active zero> --
      sig_rx_sof_n       <='1';
      sig_rx_sop_n       <='1';
      sig_rx_eop_n       <='1';
      sig_rx_eof_n       <='1';
      -- Signals set to zero <active one> --
      sig_rx_src_sw      <='0';

      -- Payload counter reset
      payload_cnt_rst    <='0';

      -- Footer counter reset
      foot_cnt_rst       <='0';
      footer_cnt_enb     <='0';

      -- Write footer data signal
      wr_footer_data     <='0';


      case STATE is

            when IDLE =>

                  -- Reset a payload counter.
                  payload_cnt_rst    <='1';
                  -- Deassert the footer write signal
                  wr_footer_data <='0';
                  -- Deassert the payload write signal
                  wr_payload_data <='0';
                  -- External Fifo disable
                  sig_ll2fifo_enable <='0';
                  -- Reset footer counter.
                  foot_cnt_rst <= '1';

                  -- Rx source and distination are ready rx sof cann de send
                  if  (sig_rx_dst_rdy_n = '0' and sig_rx_src_rdy_n = '0' and
                        Ll_Enable = '1') then
                              NEXT_STATE <= RX_SOF;
                  else
                              NEXT_STATE <= IDLE;
                  end If;


            when RX_SOF =>
```

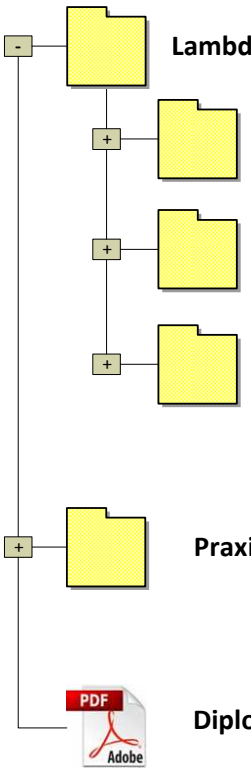
```
        if  (sig_rx_dst_rdy_n = '0' and sig_rx_src_rdy_n =
              '0') then

        -- Assert rx_sof flag
        sig_rx_sof_n <= '0';

        NEXT_STATE <= RX_SOP;
else
        NEXT_STATE <= IDLE;
end if;

when RX_SOP =>

        -- Deassert rx_sof flag
        sig_rx_sof_n <= '1';

        -- Assert rx_sop tlag
        sig_rx_sop_n <= '0';

        -- Start payload counter
        payload_cnt_enb <= '1';

        -- Assert the write payload signal
        wr_payload_data <= '1';

        -- Data can be written
        NEXT_STATE <= WRITE_DATA;

when WRITE_DATA =>

        -- Deassert rx_sop flag
        sig_rx_sop_n <= '1';

        -- Keep payload counter running
        payload_cnt_enb <= '1';

        -- Keep writing payload data
        wr_payload_data <= '1';

        -- Payload data is finished, need to send rx eop
        -- Payload Länge bestimmt durch Slave Register1(slv_reg1).
        if (sig_rx_dst_rdy_n = '0' and payload_wrt_count = slv_reg1)
        then
              -- Assert rx_eop falg
              sig_rx_eop_n <='0';

              -- rx_eof flag asserted, need to send footer
              NEXT_STATE <= FOOTER;

        else
              NEXT_STATE <= WRITE_DATA;
        end if;

when FOOTER =>
        -- Stop payload counter
        payload_cnt_enb     <= '0';

        -- Disassert the write payload signal
        wr_payload_data <='0';

        -- Start the footer counter
        footer_cnt_enb <= '1';

        -- connect rx_src_rdy to '0' during footer transmission
        sig_rx_src_sw   <='1';

        -- Footer started, need to write footer data
        if(sig_rx_dst_rdy_n = '0' and footer_count = x"4")then
```

```
                        -- Assert the footer write signal
                        wr_footer_data <='1';
                        NEXT_STATE <= FOOTER;

                -- Footer finished, need to send rx eof
                elsif (sig_rx_dst_rdy_n = '0' and footer_count = x"6") then

                        NEXT_STATE <= RX_EOF;

                        -- Deassert the footer write signal
                        wr_footer_data <='0';
                else
                        NEXT_STATE <= FOOTER;

                        -- Deassert the footer write signal
                        wr_footer_data <='0';
                end if;

        when RX_EOF =>
                -- Assert rx_eof flag.
                sig_rx_eof_n <= '0';

                -- Stop the footer counter
                footer_cnt_enb <= '0';

                -- keep the rx_src_rdy connected to '0'.
                sig_rx_src_sw   <='1';

                -- Frame is finisched, need to go to IDLE
                NEXT_STATE <= WAIT_FOR_END;

        when WAIT_FOR_END =>
                -- connect fifo empty flag to rx_src_rdy.
                sig_rx_src_sw   <='0';

                -- Waite until Ll_Enable becomes low.
                if (Ll_Enable = '0')then
                        NEXT_STATE <= IDLE;
                else
                        NEXT_STATE <= WAIT_FOR_END;
                end if;



        when others =>      NEXT_STATE <= IDLE;


    end case;
end process NEXT_STATE_DECODE;


-- Counter for payload
PAYLOAD_CTN: process (ll_clk, payload_cnt_rst)
begin
   if payload_cnt_rst='1' then
      payload_wrt_count <=(others => '0');
   elsif ll_clk='1' and ll_clk'event then
            if(payload_cnt_enb = '1' and sig_rx_dst_rdy_n = '0' and
            sig_rx_src_rdy_n = '0')then
                --Count in bytes (4 bytes each clock cycle)
                payload_wrt_count <= payload_wrt_count + x"00000004";
            end if;
   end if;
end process PAYLOAD_CTN;
```

```
--Counter for footer (Src and Dest has to be ready)
DO_RX_FTR_CNTR : process (ll_clk, ll_rst, foot_cnt_rst)
   begin
    if (ll_rst = '1') then
          footer_count <= "0000";
        elsif(foot_cnt_rst = '1') then
                  footer_count <= "0000";
     elsif (ll_clk'event and ll_clk = '1') then
        if (footer_cnt_enb = '1' and sig_rx_dst_rdy_n = '0') then
          footer_count <= footer_count + 1;
        end if;
     end if;
end process DO_RX_FTR_CNTR;

-- Just ignore the Fifo-Empty Flag during the Footer transmission
-- since the DMA-Engine will wait until source ready flag will become low.
-- The Fifo_Empty Flag is active high. If there are some data in the fifo the
flag is low.
sig_rx_src_rdy_n <= fifo2ll_fifo_empty when sig_rx_src_sw = '0' else '0';

-- Mux data depends on the FSM state.
-- Write the footer data when wr_footer_data=1 and write payload data when
payload state is active, otherwise write zero.
rx_data <= footer_data when wr_footer_data = '1'
          else fifo2ll_din when wr_payload_data = '1’
          else X"00000000";

rx_rem <= (others=> '0');

-- External Fifo Control
-- Read the fifo if rx-destination and rx-source are ready, i.e. fifo isn't
empty, and WRITE_DATA state is active.
ll2fifo_rd_enable <= '1' when (sig_rx_dst_rdy_n = '0' and sig_rx_src_rdy_n = '0'
and wr_payload_data = '1') else '0';
```

Anlagen, Teil 2

Inhalt der beiliegenden DVD



Lambda_Firmware	
LAMBDA_PPC_XPS	Enthält das XPS-Projekt des PowerPC 440-basierten eingebetteten Systems
LAMBDA_ISE	Enthält das ISE-Projekt der Lambda-Firmware
ppc_sw	Enthält das SDK-basierten Software-Projekt des PowerPC 440 Prozessors
Praxisprojekt II	Enthält das Textdokument des Praxis-projektes II sowie die zugehörigen EDK-Projekte
Diplomarbeit	Textdokument der Diplomarbeit als PDF-Datei

Eidesstattliche Erklärung

Hiermit erkläre ich, dass ich die vorliegende Arbeit selbstständig und nur unter Verwendung der angegebenen Literatur und Hilfsmittel angefertigt habe.

Stellen, die wörtlich oder sinngemäß aus Quellen entnommen wurden, sind als solche kenntlich gemacht.

Diese Arbeit wurde in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegt.

Hamburg, den 31.10.2013

Sergej Smoljanin